to Edith

Abstract

Parallel **I**nput/**O**utput **T**emplates, PI/OT, is a novel, top-down, high-level approach to parallelizing file I/O. Each parallel file descriptor is annotated with a high-level specification, or template, of the expected parallel behaviour. The annotations are external to and independent of the source code. At run-time, all I/O using a parallel file descriptor adheres to the semantics of the selected template. By separating the parallel I/O specifications from the code, a user can quickly change the I/O behaviour without rewriting the code. Templates can be composed hierarchically to construct complex access patterns.

While other approaches explicitly differentiate between parallel and sequential I/O in the source code, the PI/OT model is based on the familiar standard stream I/O (`stdio`) functionality. The current PI/OT model contains five templates that can be composed to express more complicated I/O patterns. A set of attributes for each template provides more expressibility for these basic template descriptions. The PI/OT model is intended to be imbedded into a parallel programming system (PPS). The Enterprise PPS was used to implement the PI/OT model.

Four sets of experiments test the performance, useability, and composability of these templates. The first set of experiments examines the performance of this top-down approach against versions implemented in an existing parallel I/O system (PIOUS). Two applications are used. These applications share the same parent-child computational parallelism, but have different I/O requirements. The first, based on a molecular docking application, is fine-grained and contains variable-sized objects which in turn contain other variable-size objects. The other application is a coarse-grained version of disk-based matrix multiplication. The experiments show that the performance of PI/OT is at least as good as PIOUS.

The second set of experiments examines the useability of the PI/OT model. The run-time behaviours of two applications was changed by modifying the parallel specifications without recompiling the applications.

The third set of experiments examines the effect of the complexity of the dynamic segmentation function on performance. The molecular docking application was used. With sufficient computational granularity, the complexity of the segmentation function does not have a significant impact on the application.

The fourth set of experiments takes the lessons learned in this work and creates a more complicated parallel version of the fine-grained docking application that has better performance than the simpler computational version.

These sample applications demonstrate the benefits of PI/OT model, both from the performance and the software engineering points of view.

Acknowledgements

Enterprise is a large team project and very little of this could be accomplished without the efforts of many graduate students and researchers. I would specifically like to thank Diego Novillo, Steve MacDonald, Randal Kornelsen, and Paul Iglinski for their contributions to this project. As well, I would like to thank Steve Moyer for his advice on the PIOUS implementations and discussions on the test results.

I would like to thank my two supervisors, Jonathan Schaeffer and Duane Szafron for their guidance and direction, as well, my unofficial supervisor, Ron Unrau, for his constructive comments while developing and documenting this research. I wish to thank Rod Johnson from Instructional Support Services for allowing me access to the undergraduate laboratory for some of the experimental work. This research was supported by research grants from NSERC and a grant from IBM Canada.

Finally, to my wife Edith and my three children Matthew, Ellen and Emily: I thank you for allowing me the freedom and opportunity to go back to university.

Contents

Tables

Figures

Chapter 1

1. Introduction

The development of parallel applications has focused on computational parallelism. Consequently, the corresponding growth in parallel input and output (I/O) implementation techniques has not kept pace. If an application is to perform parallel I/O operations, a user must explicitly differentiate between parallel and sequential I/O streams at the source code level, and often import or export files into or from specialized file systems. As well, the computational parallelism may have to be re-implemented to work with the communication system used to build the parallel I/O library. This results in a lack of portability between different operating systems, architectures, and even changes in the physical layout of the files.

Ideally, all of this should not be of concern to the user. A user would type in the command `cc -par Mycode.c`. The compiler analyses the code and creates the resultant binary, `a.out`. When a user runs the application, it adapts to the run-time environment. Although this is not feasible yet, the user can currently specify **what** needs to be done. With this information, **how** the requirements are accomplished can be the concern of the parallel programming system (PPS) and the physical resources controlled by the various operating systems. It should be possible to enter the command `cc -par WhatIWant Mycode.c` to create a parallel-aware binary file and to get the results by entering the command `a.out -par WhatIWant`. The parallel behaviour specifications are associated with the `WhatIWant` parameter.

This dissertation proposes a design for implementing parallel I/O requirements using high-level behavioural specifications (or templates) within the auspices of a parallel programming system. One of the advantages of a PPS is to shield a user from the low-level details of implementing parallel requirements. Several examples of these systems (with varying degrees of sophistication) can be found in [2, 4, 6, 10, 25, 28, 33, 46, 70, 83]. A PPS could use these parallel I/O specifications, along with its own model for describing the parallel computation, to implement the desired parallel behaviour. The PPS integrates all components for developing, compiling, running, debugging, and evaluating the performance of a parallel application. That is, the implementation of the parallelism is handled by the PPS. A user chooses the computational and I/O templates that give the best performance.

Current approaches to parallel I/O favour the use of parallel I/O libraries. These libraries offer an improvement over implementing the desired functionality using low-level functions offered by operating systems. The parallel I/O requirements are specified using a package of specially designed parallel I/O library calls (typically highly tuned to one or a few architectures). Usually, these libraries force the user to differentiate between sequential and parallel I/O streams and to specify how the data is to be subdivided, synchronized, and merged. There are a number of these libraries available that are designed for object-oriented, data-parallel, and parallel file implementations (for example, [7, 9, 15, 17, 20, 21, 24, 31, 34, 35, 37, 40, 42, 47, 48, 54-56, 58, 59, 63, 66, 75, 79, 82]).

When this library-of-functions approach is taken, it is important to note that the parallel behaviour is still directly coded into the program by a user. Any changes to the I/O or the parallel computation behaviour are reflected by modifications to the code. Thus, something as simple as integrating a new release of the I/O library could introduce errors. Since a user's code is implemented for a particular I/O library, if a decision is made to use another I/O library (possibly due to moving the code to a different system), modification of the source code is required even though the parallel behaviour has remained the same. As a side effect of experimenting with different parallel I/O access patterns or behaviours, many lines of code must be rewritten.

An alternative to embedding the parallel behaviour directly into the application is a high-level abstraction, or template, that separates the parallel behaviour from the code. Templates are intended to work within the framework of a parallel programming system. Ideally, one would designate an I/O stream as having a specific parallel behaviour and the PPS would correctly parallelize all the sequential I/O calls that use that stream. This abstraction mechanism is beneficial since:

- Parallel I/O and computational behaviours are encapsulated into an easy to understand set of templates.
- The user specifies **what** parallelism is needed while the template determines **how** the parallel behaviour is implemented. This can result in different solutions for the same parallel behaviour, depending on the underlying architecture or low-level software libraries.
- Parallel behaviour can be changed with minimal or no changes to the user code.
- Because the computational and I/O templates are integrated, optimizations between the different parallel behaviours are possible at both compile and run time.
- Templates provide a quick first-draft of a solution that can be incrementally refined, depending on a user's expertise.
- Correct parallel behaviour and implementation for the template are guaranteed.
- The performance of templates can be comparable to hand-coded solutions.

The programmer uses the PPS to produce a parallel application by supplying the sequential code for the parallel algorithm. The parallelism is described by selecting templates of predefined parallel behaviours for parallel computation and I/O and associating specific functions or variables to different templates. The PPS stores these templates separate from the user's code. The templates and the user's code are then processed by the PPS to generate code to perform the parallel behaviour. This machine-generated code is linked with the necessary run-time support libraries to generate an executable for a specific target architecture. This is repeated if more than one type of architecture is being used (different I/O implementations could be used that are transparent to the program). At run-time, the PPS is responsible for starting, monitoring, and terminating the parallel application.

For example, consider an application that has one of its I/O descriptors annotated to use a particular parallel I/O behaviour. The PPS analyses the source code for instances of the parallel file descriptor and modifies any code necessary to ensure the correct parallel I/O semantics (as defined by the template). If a user wishes to change the parallel I/O behaviour, a different template is specified and the PPS regenerates the code to implement the new behaviour. The strength of this approach is that different parallel I/O behaviours are specified by changing templates — not user code.

Szafron and Schaeffer examined the useability of several parallel programming systems [73]. They found that using computational templates to create parallel applications is beneficial. The user code is significantly reduced and the application is up and running much sooner since the templates are correctly implemented for the selected parallel behaviour. The drawback to templates is that there can be a slight performance penalty (i.e. less speedup). The work presented in this dissertation extends these results from computational templates to I/O templates and provides experimental validation.

There are two perceived disadvantages to using such a high-level abstraction mechanism. First, there is the loss of direct control by the user since a high-level abstraction is supposed to shield a user from many of the low-level details. Second, the performance of the application might not be as good as the hand-crafted application since the abstraction deals with the general rather than the specific details of the problem.

This first point is resolved by creating a base set of templates with user-adjustable attributes that can be composed into more complex behaviours. If more hands-on control is required, a user can change the attributes of the template (but not the code) to customize the application. The combination of simple base behaviours and adjustable attributes, coupled with the ability to be composed to build more intricate behaviours for greater complexity, provides a rich set of specifications for most parallel applications. The simple programming model, the short time to draft a working application, and the independence from implementation details typically outweigh the restrictions imposed by working within a template framework.

The second concern is more serious since, to many people, performance is the **only** evaluation metric. While this dissertation primarily addresses the software engineering benefits of template I/O, the performance of this system is shown to be comparable to hand-coded, tuned implementations. Since template I/O offers significant software engineering benefits, users should only consider hand-coded solutions if they are convinced that additional performance gains are possible. The possible performance gains may be offset by the cost of the additional effort required to implement, debug, and test their custom solution. An alternative approach for the advanced user could be to tune and modify the code generated by the PPS since many PPSs use source-to-source translation.

The system proposed in this dissertation is called **P**arallel **I**nput/**O**utput **T**emplates (PI/OT, pronounced pilot). It introduces a high-level, top-down approach to parallel I/O. A user is able to separate the parallel behaviour from the physical I/O specifications. Changes to either the parallel computations or the parallel I/O are not embedded in the user's source code. A source-to-source translation tool (precompiler) takes the specifications and creates the necessary modifications to the source code to create the required parallel behaviours. At run-time, PI/OT implements the parallel behaviours. Since PI/OT is intended to be integrated with the parallel computations, optimizations such as prefetching, declustering of data, or replications of data files can be done dynamically.

The rest of this chapter is as follows: Section 1.1 presents an example that illustrates some of the complexities of parallelizing I/O. Section 1.2 lists the contributions of this work to Computing Science. Section 1.3 describes the layout of this thesis.

1.1 A Simple Example

This section presents a simple example that illustrates some of the obstacles fundamental to parallelizing sequential I/O. The parallel program that is derived in this section is not an example of how the parallelization would be accomplished using templates. The example is intended to show the kind of code a user would need to provide if the parallelization was done by hand. Alternatively, it shows what kind of code must be generated if templates are used.

Figure 1-1 shows the sequential C code for this example along with a sample input file and the corresponding output file. The sequential program opens two files, one for reading and one for writing. The program reads integers from the input file, and for each integer, outputs a line to the output file that contains multiple copies of that integer. The input file consists of a series of ASCII character representations of integers, separated by new-line characters and terminated by an end-of-file marker. The output file can be viewed as a series of variable length character records separated by new-line characters.

This example is a simple one but it illustrates that the following basic considerations must be made when converting from sequential to parallel I/O:

- When a file is opened by multiple processes, an access mechanism must be specified. The three common access mechanisms are: **independent**, **shared**, or **seg-**

```
#include <stdio.h>

Parent( int argc, char **argv )
{
  FILE *fin, *fout ;
  fin = fopen( argv[1], "r" ) ;
  fout = fopen( argv[2], "w" ) ;
  while ( ! feof( fin ) ) {
  Child( fin, fout ) ;
  }
  fclose( fin ) ;
  fclose( fout ) ;
}

Child( FILE *fin, FILE *fout )
{
  int i, num ;
  fscanf( fin, "%d", &num ) ;
  for ( i = 0; i < num; i++ ) {
    fprintf( fout, "%d ", num ) ;
  }
  fprintf( fout, "\n" ) ;
}
```

Sample input file:

```
   3
   6
  12
   9
```

Sequential output file:

```
3 3 3
6 6 6 6 6 6
12 12 12 12 12 12 12 12 12 12 12 12
9 9 9 9 9 9 9 9 9
```

Figure 1-1 — An example program with sample sequential input and output files.

mented [23]. **Independent** access requires that each process have its own independent file pointer without any synchronization between processes. **Shared** access means that movement of the file pointer by one process affects the file pointers of the other processes. **Segmented** access implies that the processes access mutually exclusive regions of the file with their own file pointers. The user's code must be changed so that the access mechanism is explicit when a file is opened.

- For each parallel access mechanism, there are different criteria for checking the end-of-file condition and different actions must be taken to close the parallel file. These differences must be reflected in the user's code.
- Access synchronization must be specified. For example, to prevent unwanted interleaving of I/O operations by different processes, blocks of I/O statements must be identified in the code and would be considered as an indivisible or **atomic** I/O *transaction*. In addition, some synchronization may be necessary between transactions.
- The format of a file may need to be changed to support a particular parallel access mechanism.

These considerations are not intended to be exhaustive. They are given here to show that even a simple program requires extensive modifications when its I/O is parallelized. The goal is to generate these modifications automatically, using parallel I/O templates.

A natural parallelization of the program in Figure 1-1 has the `Parent` function and multiple copies of the function named `Child` each executed by its own process. Figure 1-2 shows a parallel version of the code that accomplishes this. A boldface font is used to identify changes to the code. (For clarity and brevity, the code for spawning remote processes, marshalling and demarshalling of parameters and explicit process communication is not shown.) Only two constraints are placed on the parallelization. The input file may only be read once by the user's code to avoid the duplication of work. The output of each `Child` function may not be interleaved with the output from any other. For example, it is not necessary for the 3s to be printed before the 6s. However, the 3s must appear on a separate line from the 6s.

The `Parent` process opens the input and output files using a generic parallel library function `par_fopen`. The extra parameters indicate the parallel access mode of the file (`parMode`) and the processes that will collectively share this parallel file (`parGroup`). These grouped processes that share this parallel file may be composed of subgroups within some hierarchy. This would reflect the synchronization and coherency restrictions imposed by the computational parallelism.

The `par_feof` function uses the parallel access mode set in the `par_fopen` function to determine whether the end-of-file condition has been met. For example, if shared file access was selected, then `par_feof` will be true whenever any `Child` process encounters an end-of-file condition. If independent file access was selected, then `par_feof` will be true

```
#include <stdio.h>
Parent( int argc, char **argv )
{
   par_FILE *fin, *fout ;
   fin = par_fopen( argv[1], "r", parMode, parGroup ) ;
   fout = par_fopen( argv[2], "w", parMode, parGroup ) ;
   while ( ! par_feof( fin ) ) {
   /*
    * Wrapper function to send a message to remote process
    * executing Child
    */
     par_Child( fin, fout ) ;
   }
   par_fclose( fin ) ;
   par_fclose( fout ) ;
}

Child( par_FILE *fin, par_FILE *fout )
{
   int i, num ;
   par_fscanf( fin, "%d", &num ) ;
   par_IOstart( fout ) ;                   /* Start I/O transaction */
   for ( i = 0; i < num; i++ ) {
     par_fprintf( fout, "%d ", num ) ;
   }
   par_fprintf( fout, "\n" ) ;
   par_IOend( fout ) ;                     /* Stop I/O transaction */
}
```

Figure 1-2 — A parallel version of the example sequential code.

only when the `Parent`'s file pointer reaches the end-of-file mark. In this program, that will never occur since the `Parent` never moves its file pointer. If segmented access is selected, `Parent` moves its own file pointer forward one segment at a time as it calls its children. In this program, `par_feof` will be true when it passes the last segment to a child.

The "glue" function, `par_Child`, contacts a remote process to execute the `Child` functions. This function passes the appropriate parallel file descriptors to the remote `Child` processes. Finally, the `par_fclose` function closes the file using the correct parallel access mode to dispose of the appropriate file pointers. Closing a parallel file blocks the execution of `Parent` until all outstanding `Child` processes have finished with the file.

The fundamental problem of parallel I/O programming is that multiple processes share a common resource. One of the consequences of this is that a user cannot assume a consistent I/O state between successive operations unless accesses are synchronized. Even using a parallel I/O library, a series of output operations would be interleaved unless the I/O library is informed that a succession of I/O actions are to be done as one transaction. The output operations in the `Child` function are a perfect example of this situation since the user wants all of the 3s to be output together on a line with all of the 6s on a different line. There are four approaches to solving this transaction problem. In each case, the assumption is made that a single parallel I/O operation is atomic and it is necessary to build these into larger atomic transactions.

In the first approach, each line is printed in a single I/O statement. However, since the number of output operations for each line is variable, each I/O operation will explicitly write to a memory buffer each time through the `for` loop and then explicitly write the buffer to the file at the end of the loop. That is, each process prints to a buffer using `sprintf` inside the loop (advancing the start of the buffer pointer over the previous I/O statement) and then put the memory buffer to disk using `fprintf` outside the loop to write the entire line.

In the second approach, an atomic block of output operations is explicitly identified to the parallel I/O system. This choice is presented in Figure 1-2 by the `par_IOstart` and `par_IOend` functions around the atomic I/O operation.

In the third approach, each remote process gets a block of the file to which it has exclusive access. Each process can then concurrently write its output without fear of interference. However, this approach is complicated if variable-length output records are needed, unless the block size can be easily determined in advance of using the block (either by static analysis or dynamically).

In the fourth approach, each remote process writes to a local scratch file; after the transaction is finished, the file contents are returned to the parent to be integrated into the master file. This approach is similar to the first approach, except that it is intended to be managed by a parallel I/O system instead of being the explicit responsibility of the user.

In addition to a mechanism to delimit atomic I/O transactions, it is often necessary to specify the synchronization of I/O primitives themselves. For example, the `par_fclose` function cannot actually close the file until all `Child` functions have finished with the file. Code must be written in the `par_fclose` function to perform this synchronization.

Sometimes the structure of files must be changed to support a parallel access mode. For example, if segmented access to the input file is desired for the program in Figure 1-2, then fixed length records would be easiest to support. One way to do this would be to store the integers in binary format instead of ASCII format. Alternately, if ASCII format is necessary, then a fixed number of characters must be specified for each integer. This has the disadvantage of restricting the range of the input data, say from -999 to 9999, if four characters are used. Similarly, if segmented access to the output file is used, a fixed size line for the output file would be required as it is difficult to predetermine the size of a particular line or file segment. Consequently, the file would be padded with blank or null characters.

It is clear that even a very simple program requires extensive modifications to parallelize the I/O operations. As Chapters 3 and 4 will show, templates provide a good mechanism for generating much of this tedious code automatically, while Chapter 5 demonstrates that the template approach can provide reasonable performance.

1.2 Contributions

This thesis makes the following research contributions:

- This work demonstrates that parallel I/O specifications can be separated from the sequential functions. That is, PI/OT keeps the standard sequential interface used for invoking any I/O operations in the user's code and describes, independent of the user's code, **what** parallel I/O behaviour(s) are needed. At compile and at run-time these specifications are used to identify and implement **how** the parallel behaviours will interact with the application and its environment.
- By separating the I/O and computational parallelism from the sequential code, it is possible to support optimizations and adaptive behaviours by using the captured knowledge of all the parallel behaviours, both at compile time and at run-time.
- This work demonstrates that there are significant software engineering benefits to I/O templates including: less code, rapid prototyping, and fewer errors. As well, it demonstrates that I/O templates can generate code whose performance is comparable to hand-coded parallel I/O.
- By identifying the components that interact between the computational and I/O parallel behaviours, this work shows how optimization and run-time characteristics are handled in a more automatic and efficient manner.
- This work provides a contribution towards automatic parallelization by the successful separation and integration of the various parallel behaviours.

1.3 Document Structure

This chapter outlines the motives for this research and describes the goals and scope of the thesis. The example (Section 1.1) illustrates the complexity of parallelizing the computational and I/O aspects of even a simple application. Chapter 2 provides a summary of the related work used to develop the model presented in this dissertation. Chapter 3 presents the parallel I/O model used for PI/OT. Chapter 4 discusses the implementation of the model in general terms as well as a specific implementation within the Enterprise parallel programming system. Chapter 5 compares the performance of the Enterprise version of PI/OT against the equivalent implementations using PIOUS [57], a low-level parallel I/O system. This chapter also explores the composability and useability of the templates to construct more complex I/O patterns with two different parallel computational models. Chapter 6 describes some user and system optimizations and extensions that are possible, along with future research directions. Finally, Chapter 6 summarizes the contributions of this work and presents conclusions.

Chapter 2

2. The Current State of Parallel I/O

This chapter presents a summary of the body of work that was used to inspire the specific approach to parallel I/O proposed in this thesis. In Section 2.1, some of the specific terminology used in this dissertation is defined. The balance of this chapter presents a review of the current state of parallel I/O research as it relates to this dissertation.

The current state of parallel I/O research can be divided into three parts: characterization, actual parallel I/O systems, and integration with a parallel programming system. A problem must be characterized and studied before any solutions can be examined. Parallel I/O systems are derived from the results of characterizing problems. How are these I/O systems integrated into the computational mechanism? Or, are they stand-alone parallel file systems? If a given system has chosen to ignore the UNIX interface and sequential file system, what must the user do in order to cross the boundary between parallel and sequential I/O? Parallel I/O solutions need to be integrated into the existing parallel computational solution. That is, I/O and computation must be considered in tandem. How easy is it for a user to make changes to either the computational or I/O parallelism without making significant changes to the other? This is an important question if the system is to react dynamically to changes in the network, processors, and file-systems.

Section 2.1 introduces specific definitions to some of the terms used in this work. Depending on the reader's background, a specific term may have different meanings. The intent of this section is clarify understanding by providing a single definition. Section 2.2 summarizes the characterization of parallel I/O as well as some attempts to parallelize the I/O in various real applications. Section 2.3 characterizes the current state of parallel I/O libraries. Section 2.4 discusses the use of a separate file system to efficiently implement a desired parallel behaviour. However, utilizing the existing sequential file system by coordinating access may be equally efficient and has the added benefit of not requiring the duplication of files or pre- and post-processing of the data files. Section 2.5 examines the object-oriented approach to parallel I/O. Section 2.6 describes the current state of parallel programming systems. The complexity of implementing parallel I/O implies that there must be cooperation with parallel computational systems. How these existing parallel computational systems support parallel I/O is examined. Section 2.7 presents four issues for parallel I/O. Finally, Section 2.8 provides a summary of this chapter.

2.1 Terminology

Templates have been used to express parallelism in many parallel programming systems (PPS). For example, templates have been used to express the computational parallelism in Enterprise [70], HeNCE [5], and P^3L [3] and to define data parallelism in High Performance Fortran (HPF) [41]. Templates are pre-defined behaviours with a well-defined interface that allow the user to express to the PPS **what** is needed while the PPS can determine **how** to implement the behaviours. Typically, templates are used to express simple behaviours that can often be composed to represent the complex behaviours of an application. The well-defined interface allows the PPS to interact with the different templates to determine how exactly the complex behaviour is implemented.

Parallel templates should not be confused with C++ templates. Although TPIE [82] and Mentat [33] use C++ templates to express parallel behaviours, in this thesis, templates do not imply a C++ language binding unless explicitly noted.

I/O and computations are inextricably tied in an application. The traditional view of the temporal ordering of data input, computation, and data output must still be respected when entering the parallel domain. This ordering may be necessary for *program correctness*. To a user, the order in which statements are executed often determines whether the program performs correctly. If the I/O operations in a parallel program must occur in exactly the same order as the equivalent sequential program, the I/O is defined to be in *sequential* order.

However, in parallel programs, users may specify several levels of acceptable behaviour depending on the application's requirements. These levels are due to the degree of concurrency now available to the application. Recall the example program given in Section 1.1. The sequential version of the program opened an input and an output file, then repeatedly read in integers and output variable length character strings until the input file was exhausted (EOF was reached). When the application was parallelized, one of the constraints was that the entire output line for a given input was to be considered as one atomic I/O operation even though multiple I/O operations were needed to create it (the `for` loop). However, the order of the lines themselves was not important. In this case, the sequential ordering was relaxed to a *serialized* order in which atomic blocks could be output in an arbitrary order.

The input file had the constraint that the data must be read once, regardless of how many processes accessed the file. By segmenting the file, many processes could independently read different parts of the input file concurrently. Such input is called *chaotic* as no process ordering is needed to access a file segment.

All of these input and output access patterns are correct according to the user constraints. However, the implementation of these patterns is complicated by the computational parallelism and the run-time environment. For example, the number of cooperating processes and the physical location of the data files will affect the overall performance. From a parallel I/O viewpoint, the access patterns (chaotic, serial, and sequential) can be viewed as a level of "correctness" since they define progressive restrictions on I/O behaviour.

Sequential correctness is the most restrictive access pattern with I/O operations proceeding in the same order as the sequential application. Significant synchronization is required, with a corresponding reduction in concurrency. Serial correctness implies that there are blocks of work to be done but that the order of the blocks is not important. However, each block of work has its own internal view of correctness that is irrelevant outside the block. For example, some blocks may be sequential and others might be serialized. The chaotic level is a complete relaxation of ordering where the program executes with minimal (if any) synchronization.

Regardless of the level of correctness, multiple concurrently executing processes require a user (or some intelligent agent) to implement synchronization mechanisms to ensure correct parallel behaviour. Example mechanisms are: **barriers** to ensure all processes complete a certain task; **rendezvous** to coordinate senders and receivers; and **semaphores** to indicate exclusive access. The user is responsible for specifying the desired parallel behaviour and the level of correctness for the application. However, the PPS is responsible for implementing the synchronization. The PPS resolves the different parallel requirements of the application to produce an overall parallel behaviour. These requirements include the computation, the I/O, and any global or shared memory.

From a parallel I/O perspective, there are two aspects to an application's parallel behaviour — static and dynamic. The static (or compile-time) component identifies all possible cases where a parallel file pointer is used, determines which virtual processes share the file, and resolves the boundaries defining a given I/O transaction. The dynamic (or run-time) component decides which physical processes share the parallel file, how much optimization (for example, prefetching or caching) can be done, and exactly how much of the file is shared, locked, or modified.

Computational parallelism has an effect on the I/O behaviours. This can be seen in the simple case where an application stays in a loop that inputs data, performs a computation and outputs data until some exit condition is met. If the loop is parallelized to use concurrent processes, a user may want to avoid reading the same input data more than once and may demand no interleaving of output lines. Synchronization and coordination of the input and output streams are needed. If an application splits a computation into several parts all running concurrently, the layout of the data in the file may require the user to impose barriers to prevent the application from reading or writing to the wrong part of the file.

Changing the parallelization behaviour of the I/O can also affect the efficiency of the computational parallelization. Whether these effects are positive or negative, they cannot be ignored. If the parallelization details can be separated from the computational requirements, the parallel details can be separated from the sequential I/O calls. The overall motive is to ensure that positive results are possible.

2.2 Parallel I/O Characterization

The basic types of parallel I/O are still the same as when Crockett [23] characterized them — global, segmented, and independent. However, optimizations for specific architectures and algorithms can be used to create specialized solutions (for example, strided interfaces, and disk striping). Nevertheless, integration of the parallel computation, run-time support libraries, architecture, and network characteristics are essential to provide a good general parallel I/O solution. A clearer understanding of these relationships permits the programmer to create efficient parallel applications.

Various papers have discussed the optimization of an application's I/O. Most of these papers concentrate on specialized architectures (such as Hypercube, CM-5, and SP-2) and their associated custom I/O software. The network configuration is largely ignored except to note that it should be as fast as possible, dedicated only to the application, and that fully connected processors are desirable. The capacity and speed of the communication network are perhaps the dominant considerations in determining the best solution for a given application. That is, a slower network solution can trade the speed of locally cached data files against the complexity of ensuring cache coherency. Alternatively, a large number of concurrent processes sharing access to a given file can make the cache coherency solution too expensive.

I/O optimization can be approached from several directions. One way is to examine traces of "real" applications running on existing systems [22, 60, 67, 76, 77]. From these traces, a file system designed to optimize parallel I/O can be developed or tuned for the given system and application suite. Another approach is to create a set of test applications to characterize the best I/O configuration for a given machine [32]. The user can then engineer the application to take advantage of a particular configuration. A more general approach takes an algorithm and documents the steps necessary to maximize throughput, irrespective of the architectural platform.

There are three problems with these approaches. The first is to get all of the users to cooperate with the study. If a computational platform allows users to run their applications concurrently, uncooperative users could contaminate the traces by consuming some of the platform's capacity in an unknown fashion. The second is to ensure that there is enough variety in the applications to draw useful generalizations from the study. The third is to determine whether the stability of the machine(s) and software available impose constraints on the potential solutions. Typically, the "best" solution is a compromise between the existing software and hardware and the amount of the programmer's time available to develop an acceptable solution. As well, the run-time environment may indicate that the optimal algorithmic solution with the system under heavy load is not optimal when the system is lightly loaded.

For example, Nieuwejaar and Kotz [60] studied traces of existing applications on various parallel systems. From their data, they determined that regular steps or strides through data are common. Consequently, they have developed strided and nested-strided interfaces [58] which have led to the Galley File System [61] and the disk-directed I/O proposal for parallel I/O [47].

Womble *et al.* [85] examine the LU decomposition algorithm executing on a Paragon and an nCube. One of their conclusions is that having background I/O to overlap computations is an important component of a parallel file system. As well, a partitioned file system is important for high-performance.

Acharya *et al.* [1] chronicles the steps needed to parallelize the I/O in four applications that have overall I/O requirements of between 75MB and 200GB on an SP-2 with a high-capacity I/O system. Three of the applications were tuned to get much better throughput. They found that complicated I/O interfaces, such as strided I/O requests, were not always the best answer. The need to modify code, to use local disk storage where possible (avoiding congestion on the network), and to have the knowledge of future I/O requests (when to prefetch) are sufficient to give significant improvement to throughput and speedup.

These four applications were tuned using the Jovian-2 parallel I/O system which, unlike its predecessor Jovian [7], is a multi-threaded client-server system with a simplified interface similar to the POSIX `lio-listio` interface [43]. This allows multiple I/O requests to be issued with one call. The rewrite of Jovian was indicated after the collective I/O interface did not work well with real applications.

Different researchers have drawn different conclusions from their characterization efforts. Conclusions differ according to the extent of the modifications to the user's code necessary to "simplify" the parallelism and according to the nature of the applications being parallelized. Simplifications are certainly useful, but at what cost to developing real applications?

Characterizing well-understood parallel applications and algorithms under controlled conditions facilitates development of optimization techniques. However, the study of the I/O complexity of a task requires that many components be held constant. For example, having a homogeneous architecture and network, or generating the parallelism with explicit knowledge of future requests, is not always possible. Parallel programming systems try to shield some of this heterogeneity from the user. Can the abstraction techniques used by these PPSs be utilized by parallel I/O templates?

2.3 Parallel I/O Interfaces

Parallel I/O interfaces can be roughly divided into two groups: virtual parallel file systems (which are addressed in this section) or real parallel file systems (Section 2.4). Virtual parallel file-systems reside within the conventional UNIX file system. Within these divisions, there is the library approach of separate function calls (as discussed here) and the object-oriented approach (Section 2.5). Many systems leave the user to specify the desired parallelism and to coordinate the synchronization. This puts the user in the position of encoding the I/O parallelism directly into the application. Changes to either the computational or I/O parallelism may then require extensive code modifications.

Four representative virtual parallel file systems using this library-of-functions approach are presented in the following sub-sections. They are PIOUS, MPI-IO, CUBIX, and CHIMP.

2.3.1 PIOUS

Parallel **I**nput/**OU**tput **S**ystem (PIOUS) [57] provides parallel I/O operations for processes using PVM communication primitives. The basic principles of PIOUS are that it uses an asynchronous model with independent individual servers, data declustering for scaleable performance, and network transport and native file system independence to enhance portability. Each client uses a special library of functions to translate file operations into service requests with the various PIOUS data servers.

PIOUS has a single service coordinator that initiates major system events such as opening a file by a client. The service coordinator deals with the meta-data and not the actual file access. Each processor involved in the declustering of the data files has a data server that acts independent of all others, enhancing scalability. Ideally, the servers access local files on disks physically connected to the processor, but a network file system does not pose a problem. The server does not interpret the byte stream, but leaves that up to the user. Sequential UNIX files must be imported into the PIOUS system before any of the parallel I/O functions can work with the data. Similarly, after the application is finished, the PIOUS file must be exported back to the sequential UNIX file system before processing by non-PIOUS applications.

The parallel I/O operations are done as transactions to provide sequential consistency for the user. There are two different transaction types: stable and volatile. Stable transactions guarantee that coherency is preserved in the case of a system crash. Volatile transactions do not guarantee coherency if a system crash occurs, but they do provide high performance.

PIOUS is based on a parallel access object, `parafile`. Each `parafile` is logically one file, but it is composed of physically distinct segments. The segments are set at the time of creation and cannot be changed. The `parafile` is globally named within the PIOUS system. PIOUS only supports an uninterpreted byte stream. Where the I/O is done in an environment consisting of heterogeneous computers, the files must be stored in universal data representation (UDR) format. Work is being done on storing record formats for these types of files.

An I/O operation is usually considered as one transaction. For more complicated transactions, the user can explicitly start, end, and abort a transaction composed of multiple I/O operations.

There is a clear separation of parallel and sequential I/O in PIOUS. The user must explicitly encode all the parallelism into the code. This is consistent with the PVM philosophy of providing a basic set of tools for the user to construct parallel applications. A drawback to this library-of-functions approach is that changes to the computational parallelism in the PVM application are not recognized by PIOUS.

2.3.2 MPI-IO

MPI-IO [20] started as a separate entity from MPI [83] but has since been integrated into the MPI-2 [53] proposal. The MPI-IO working group decided to provide a complex interface consisting of more than 45 I/O related functions. This complexity reflects the desire of the group to keep each function simple and focused on one parallel I/O task. Initially however, this plethora of choices appears daunting to the user.

The MPI-IO system supports two kinds of parallel I/O operations — independent and collective. The coordination of a file is limited to the members of the communication group used to open the file. An independent I/O operation does not coordinate with any other members of the communication group. However, if the user selects a collective I/O operation, all members of the communication group must participate. The completion of the call by one process does not mean all processes have started or completed the call. Each process is free to intermix individual or collective I/O operations.

The MPI-IO system maintains two file pointers. One file pointer is local to the process. The other is global and is shared between all members of the communication group. There is a collection of I/O routines which use the shared file pointer. Use of the shared file pointer leads to the serialization of multiple calls with non-deterministic results.

In MPI-IO, the contents of a file are specified by an MPI derived datatype — an `etype` list. The `etype` list is a description of the fields of data stored at specified offsets. Thus, "holes" in the data stream are possible. When opening a file, the user specifies an absolute `displacement` in bytes from the beginning of the file. Subsequent access is defined by the two `etype` lists: `filetype` and `buftype`. The `filetype` describes the disk layout of the file either partly or completely. The `buftype` list describes the layout in the application's memory buffer for each read and write operation. The `displacement`, `filetype`, and `etype` can be changed later to access different parts of the file. This may appear confusing, but it makes sense from a parallel programming viewpoint. A user defines different I/O behaviours or data views within the code. There is no support for the concept of transactions in MPI-IO. Each I/O function is considered atomic. A developer is expected to use the general MPI system to synchronize if an I/O operation takes more than one function to complete. It is considered an error if a file is opened for shared or collective operations by individual processes using different disk layouts.

At the time of writing, there are two alpha releases[1] of MPI-IO but they are incomplete and are based on earlier releases of the design document. It is not clear when a more robust and complete version of MPI-IO will be released, especially since it is now being integrated with MPI-2.

The main drawback of MPI-IO is that it is attempting to create a standard that encompasses C, C++, FORTRAN, and FORTRAN 90 language bindings. Each of these languages approach I/O differently. C views I/O as a stream of bytes and imposes structure from within the application. FORTRAN has fixed or random sized records. C++ (object-oriented) has each object interpret a stream of bytes. Normally, complex objects tell their sub-objects to read in data from the disk. This leads to the I/O being distributed throughout the code and having finer granularity. By trying to create a standard for all, it is likely that only a common unsatisfactory subset will emerge. It is not clear that leading research by developing a standard is the best approach at this point. Clearly, a standard will be useful — eventually. However, this area of research is still in a state of flux and standards at this time would likely inhibit the introduction of alternate solutions.

2.3.3 CUBIX I/O

The CUBIX I/O model [29, 69] defines two types of streams. The first is the traditional single stream mode while the second addresses the concurrency found in parallel applications. The user can explicitly switch a file stream between **single** and **multiple** mode. The CUBIX model is based on loose synchronicity and rank ordering of the processors. This ranking provides an access ordering to the file. There are two access methods. In **single** mode, all clients execute the same file function with identical data and only one arbitrarily selected client's data is transferred. The **multiple** mode occurs when all clients execute the same file function with differing amounts of data. The order of transfer to or from the file is determined by the node identifier ranking (lowest to highest).

One limitation of the CUBIX I/O model is that all processes must execute the same I/O functions at the same time and block until a lower ranked node has released the stream to proceed. Reordering of the data file may be required. The example shown in Figure 2-1 illustrates this point.

---

[1] IBM: http://www.research.ibm.com/people/p/prost/sections/mpiio.html and
NAS: http://lovelace.nas.nasa.gov/MPI-IO/pmpio/pmpio.html.

```
ParFunc( )
{
   int i, j ;
   scanf( "%d", &i ) ;
   scanf( "%d", &j ) ;
}

Input: 1 2 3 4 5 6 7 8
```

Sequential		**Parallel (4 processes)**	
i	**j**	**i**	**j**
1	2	1	5
3	4	2	6
5	6	3	7
7	8	4	8

Figure 2-1 — Example for CUBIX I/O.

In this example, the function, `ParFunc`, is replicated four times (i.e. there are four processors executing `ParFunc` concurrently). In the sequential version, this function is called four times. The table shows the values of the variables `i` and `j`, depending on whether the function is run concurrently using CUBIX I/O or not. All four processes execute the first `scanf` and, once done, all four do the second `scanf`. To get sequential results in the parallel version either the data file must be reorganized to reflect the parallelism, or the two read statements must be consolidated into one read operation.

Express [27] uses the CUBIX model to help the user partition data files [63]. There are three I/O abstractions: one process for multiple channels, multiple processes for multiple channels, and multiple processes for a single channel. Express depends on the user to explicitly insert additional I/O function calls for the parallel behaviours. The user defines how a group of processes will partition the data file. By using the Express functions to define the partitioning, the I/O subsystem re-aligns itself with the processor mapping. The end result is that the user's conventional I/O calls operate normally.

LAM [62] is a distributed memory, multiple-instruction-multiple-data (MIMD) programming and operating environment for a network of heterogeneous UNIX workstations. It is a subset of the TROLLIUS [11] system that provides parallel support for dedicated processor systems. LAM supports parallel I/O based on the CUBIX file access model. LAM differentiates I/O insofar as there are separate I/O functions for CUBIX and non-CUBIX (UNIX) operations. The UNIX version has each process write directly to an I/O stream with no synchronization. The LAM system supports MPI, PVM, and its local message passing functions. The user is still expected to write parallel code by using the low-level library functions.

2.3.4 CHIMP

Common **H**igh-level **I**nterface to **M**essage **P**assing (CHIMP) [18] is a parallel programming environment similar to LAM. The **P**arallel **U**tilities **L**ibraries (PUL) [10] are built on the CHIMP base. Two of the relevant libraries support parallel I/O and parallel data management. There are two PUL utilities for parallel I/O operations and two for parallel data management.

The first utility is a **G**lobal **F**ile utility, PUL-GF [13], which provides access arbitration for a group of processes with a common shared file. GF provides the C `stdio` functionality of structured and unstructured access to shared files. There are four modes of access. Two modes, **single** and **multi**, behave similarly to the CUBIX [63] model discussed in the previous section. The **random** mode allows processes to independently access arbitrary data using a global file pointer. The **independent** mode gives the processes a local file pointer. Modes can be changed dynamically. The current implementation is a client-server architecture providing non-blocking I/O operations that let computations overlap with the I/O operations.

The second parallel I/O utility is PUL-PF [15], or **P**arallel **F**ile system. PF is intended to provide a transparent, efficient, and portable interface to parallel disks. The developers of

PUL-PF feel that the conventional UNIX byte stream model is obsolete. An application data structure is used to control file access operations. The distribution of the file data is done at a user-defined record level using an I/O atom of possibly variable length. There are data distribution strategies available to the user that permit the optimization of I/O depending on the problem and architecture available.

PUL provides parallel data management that addresses the performance of applications with regular local operations over large data sets, such as computational fluid dynamics or seismic data processing. Data is distributed and processed according to the *owner computes* rule. That is, the owner is responsible for boundary data consistency. The PUL-RD [14], or **R**egular **D**omain decomposition utility, has operator stencils (similar to templates) to calculate the inter-process communications between boundary updates. The user has the option of blocking on I/O or overlapping computation with I/O operations. The PUL-SM [80], or **S**tatic **M**eshing utility, supports irregular mesh-based problems that suffer from load imbalance and need dynamic reconfiguration. The SM utility supports two and three dimensional meshes, ensures consistency of data boundaries, and provides data migration.

CHIMP and PUL are based on MPI. It is interesting to note that a user can abstract application requirements by assigning parallel templates or meshes to data sets. However, the user's code still contains the explicit parallelism. As well, the developers have abandoned the traditional UNIX byte stream model. This forces a user to redesign an application before using this system if the application used UNIX semantics. There is also the differentiation between a general network file system and a specialized parallel file system. Crossing this boundary is neither transparent nor trivial.

2.4 Parallel File Systems

Section 2.3 looked at I/O libraries that provide a virtual parallel file system. However, a real parallel file system is another alternative. Five representive systems are presented here. The last of these systems is more than a parallel I/O library but less like a parallel file system.

VESTA [21, 26] uses a two-dimensional file layout and a client-server structure to control accesses to parallel files. It defines a **basic striping unit** (BSU) with I/O processes managing multiple BSUs. By managing access to stripes, concurrency of I/O operations provides improved throughput.

The GALLEY file system [59] enhances the VESTA approach by providing a three-dimensional view of parallel files. A parallel file is divided into a series of distributed **subfiles** where each subfile is further subdivided into a number of **forks**. A fork is similar to a familiar sequential UNIX file. This is well suited to dynamic record sizes and application-specific clustering of data. GALLEY provides three access mechanisms to the data: a simple striding, nested striding, and an unstructured interface.

The **P**ortable **P**arallel **F**ile **S**ystem (PPFS) [42] provides a portable parallel I/O library to allow a user to control file caching, prefetching, data layout, and coherence policies. It provides a number of predefined policies but does allow a user to define layout, access-patterns or new prefetching policies.

The **VI**rtual **P**arallel **F**ile **S**ystem (VIP-FS) [35, 36] is a layered approach to parallel I/O. The local file systems are connected by I/O processes that cooperate with the **V**irtual **P**arallel **F**ile (VPF) layer to provide a single file image to the interface. A user can access files by conventional UNIX calls such as `open` and `lseek`. Each process in the distributed parallel application has complete access to the file. A user is responsible for coordinating file access. Alternatively, the parallel file can be partitioned and mapped to the various distributed processes using the specialized VIP-FS function calls.

The Panda parallel I/O system [17] is designed for single-program-multiple-data (SPMD) scientific applications. It uses an HPF distribution schema for the data arrays with a server-directed I/O architecture. This server-directed approach allows a more controlled gathering of data chunks to take advantage of the lower cost of larger I/O operations.

To summarize, all of these systems have a separately defined interface for parallel I/O. Some of the systems provide an interface that permits the user to work with the familiar UNIX I/O functions to access data. However, to stripe or distribute the data requires explicit calls to specific parallel file system functions.

2.5 Object-oriented Parallel I/O

Many object-oriented applications could benefit from parallel I/O. Object-oriented applications do not necessarily have the same I/O characteristics as a traditional high performance computing (HPC) application (like computational fluid dynamics or systems of equations). Typically, objects define their own I/O so that complex objects rely on the I/O operations of internal objects. This decentralization of I/O requires more synchronization and coordination between processes and the file system(s). For example, one approach could cache I/O blocks into local memory to amortize the cost of the smaller I/O operations. The task of the PPS is to identify and extract or merge the correct data block.

Three representative object-oriented parallel I/O systems are presented.

The Mentat group [33] has implemented the **E**xtendib**L**e **F**ile **S**ystem (ELFS) [34]. ELFS is designed so that a user implements a file system optimized on a class by class basis. Prefetching and caching strategies, as well as striping and partitioning across multiple physical devices, are supported. The consistency semantics for a given class may be relaxed from the strict UNIX semantics of immediate visibility after a write operation. It is both an advantage and a disadvantage of ELFS that a user must define and extend the parallel I/O behaviours.

A **T**ransparent **P**arallel **I**/O **E**nvironment, TPIE [82], uses C++ to implement parallel I/O access patterns. A user builds a stream of data stored on disk. Various access patterns are pre-defined for a user to associate with a file. The intent of TPIE is to abstract the I/O details, leaving a user to specify only the required I/O behaviour.

The **H**urricane **F**ile **S**ystem (HFS) [48] is the parallel file system for the Hurricane distributed operating system [81]. This custom file system allows a user to build hierarchies of data objects that reside in memory or on disk. Because HFS is designed to work with a supportive distributed operating system, many of the data management routines (such as cache management) are part of the operating system. This leaves a user free to concentrate on higher level parallel I/O concepts.

These systems still require a user to encode the parallelism into the application using explicit parallel functions. Also, the specialized operating system providing support for the file system is intended for research and is not widely available.

2.6 Parallel Programming Systems

Parallel programming systems (PPS) are essential for developing parallel applications. Since I/O is an integral part of any application, some means of integrating and coordinating I/O and computational parallelism is needed. This section looks at several high-level PPSs and several lower-level communication libraries used to develop parallel applications. Often, a PPS or an I/O library has developed parallel I/O systems based on an underlying communication or parallel computational model.

Parallel programming systems can be divided into two groups. The first group uses some form of abstraction to allow the user to specify the parallelism at a high level. PAMS [6], HeNCE [4], Mentat [33], Enterprise [70], and High Performance Fortran

(HPF) [41] are examples of these systems. Typically, these systems use a compiler tool that processes a user's source code along with the selected abstractions to produce a parallel binary.

HeNCE and Enterprise use a graphical interface to let the user describe the parallelism by means of templates or pre-defined behaviours. There has been no direct effort to support parallel I/O in either system. HPF has compiler directives to distribute the data. PAMS requires the user to define the parallelism by means of structured comments. This hides annotations from a conventional compiler so it can build sequential applications with the same code. Mentat extends the C++ language through added key words. The user defines `mentat` parallel classes. By taking advantage of inheritance in C++, Mentat implements parallel communication with the marshalling of data handled automatically.

The **P**arallel **A**nd **S**calable **S**oftware for **I**nput-**O**utput (PASSION) system [75] is a compiler and run-time library for HPF applications. A user provides directives about data distribution, and the compiler manipulates and transforms the source code to map the out-of-core data to disk. Because all the information about the parallel computations and I/O requirements are available, techniques such as prefetching and collective I/O operations can be efficiently implemented. This system does not address the problem of file access *per se* but it does show the effectiveness of having sufficient information to make informed optimizations.

A data-parallel I/O system, Stream* [55], lets the user keep a parallel programing view (C* [38, 78]) and familiar C file routines. Hints are placed in the source code to distribute the I/O by specifing a *shape* (a physical file layout) for the I/O and identifying parallel variables using that shape. These hints enable the system to partition a file for data-parallel SIMD and MIMD computations. The parallel files have an associated meta-file that describes the I/O parallelism. While this system is intended for data-parallel computations using a specific parallel programming language, it does maintain standard I/O system calls. It gives hints to the run-time system about the desired parallelism while still giving reasonable performance.

The second group of parallel programming systems provides libraries of "primitive" functions to let the user encode the explicit parallelism into the application. MPI [83], PVM [30], p4 [12], and LAM [11] are examples of these systems. The user is responsible for all aspects of the parallelism including launch, communication, and shut-down. Typically, the user writes code using a library of supplied functions and adds the appropriate library at link-time.

The library-of-functions approach is complicated since a user is responsible for using the library correctly. A user not only needs to develop a parallel computational framework but also to define a parallel I/O model and to integrate the framework and model in the run-time code. A user ends up developing a series of "glue" functions that use the parallel computation information to implement the desired I/O behaviours. The disadvantages are the potential for error and the cost of learning the system.

The systems that use a compiler to process the user's code could be modified to provide the necessary analysis of the user's code for parallel I/O behaviours. While this is fine for static analysis, each system would need to develop a parallel run-time I/O behaviour and integrate it with the parallel computational behaviour. This is typically a one-time cost.

With the exception of Enterprise, the above systems differentiate between the parallel and sequential behaviours explicitly in the source code. None of the systems use templates to express parallel I/O behaviour *separate* from the I/O function calls. That is, none of them allow a user to develop an application using the familiar sequential I/O functions and specify the parallelism separately.

2.7 Four Issues

Despite the apparent simplicity of the computational parallelism found in the example application in Chapter 1.1, there are four issues for the application I/O that a user must address when moving to the parallel domain. These issues are not the superficial ones from the example of opening or closing of a file, sharing file pointers, and atomicity of I/O requests, but are a deeper and more fundamental set of issues.

The first issue is a physical or operating system (OS) issue. The support offered by conventional mainstream operating systems is for distributed or single process applications, not parallel applications. There is a difference between a distributed application and a parallel one. An airline reservation system is an example of a distributed application while the example in Chapter 1.1 is a parallel application. These operating systems, while providing I/O tools to aid a parallel programmer, do not directly support parallel applications.

A user submits an I/O request to the OS as a function call. There is no direct contact between the process and the disk. Typically, many processes are active on a given processor. These processes are themselves sharing an OS kernel data buffer consisting of several pages of the physical file. If many processors are accessing the same external data object (for example, a distributed or parallel application), all these distinct and independent kernel buffers must be coordinated and synchronized. If the OS does support parallel files directly, the PPS or user must explicitly supply the coordination and synchronization functionality before allowing the OS to complete the I/O operation. This may mean that the process that issues the I/O request may not be the same process that actually does the I/O operation.

The second issue is the matching of the application to the parallel I/O model. A number of papers have studied parallel I/O characteristics. One approach is to have a typical application suite tuned to an existing architecture and file system through analysis of physical traces of the I/O calls. Another approach is to document the steps necessary to parallelize the I/O for a series of applications to run on a particular system. From these studies, optimizations such as disk-striping, prefetching, and strided interfaces have been developed.

The third issue is matching the computational parallelism and I/O parallelism. A significant collection of parallel programming systems exist that abstract the parallelism to a lesser or greater extent. If a parallel I/O operation is to successfully take place, a number of unknowns must be determined. The number of cooperating processes, which processes are actually doing the I/O, how the data is mapped to the physical file layout (overlapping or adjacent page boundaries), and what data is needed in the near future at a given process (prefetching) are just some of the information that can be supplied to optimize parallel I/O operations.

The fourth issue is the approach taken to parallelizing I/O. The consensus appears to be that it is preferable to have a separately defined parallel I/O interface with a distinct application program interface. This implies that the parallelism is now embedded in the application. This is a poor choice from a software engineering viewpoint since there are many different parallel I/O interfaces. With a distinct parallel I/O interface, changes to the I/O runtime environment or switching to a different library may require user intervention. This intervention does not reflect any changes to the underlying parallel I/O behaviours but addresses only the mechanics of **how** to implement the desired behaviours.

The approach of differentiating between parallel and sequential I/O streams both complicates and simplifies the programmer's coding strategy. It simplifies the problem since only the parallel I/O is converted. The complication arises because a user must choose which files to parallelize and then decides on the parallel I/O model and its implementation (library) before starting to write code. Templates would allow the user to switch between sequential and parallel I/O at any time, independent of the code. This leads to more portable and maintainable code.

A template approach can use any low-level parallel I/O implementation that supports the expressed parallel behaviour of the template. The basic types of parallel I/O are still global, segmented, and independent. **How** they are implemented, either as a library for a specialized file system, as an operating system module, or even as hardware, is strictly a matter of efficiency. The interface to the user must be simple but flexible enough to express **what** parallel behaviour is desired for a specific application.

2.8 Chapter Summary

At present, the idea of separating the description of parallel I/O behaviour from its sequential counterpart has not been exploited. Rather, all approaches to date have involved the development of a separate interface to exclusively handle the parallel I/O. The user is required to explicitly encode the desired parallel behaviour into the application. Some work has been done to separate the computational parallel behaviour from the sequential code. This allows a user to easily change or test different parallel versions without major code revisions. This chapter outlined the motives for extending this separation technique to the I/O component of a parallel application.

Chapter 3

3. Parallel I/O Model

In Chapter 2, various approaches to parallelizing I/O were outlined. This chapter presents the parallel I/O model central to this dissertation. The purpose of this model is to separate the I/O parallelism from the physical I/O. That is, a user specifies **what** needs to be done using these templates but not **how** to accomplish the task. The model is implemented as a series of templates representing parallel I/O behaviours. By modifying attributes of the template behaviour, the user is able to customize the template to the application in a code independent manner. As well, more complex I/O behaviours can be created by inheriting a caller's I/O constraints and imposing them on top of the current I/O behaviour.

The I/O model is only one part of the overall parallel programming system. While parallel I/O does have special needs, it must be recognized that there exists an interdependence with the computational parallelism and the physical system running the application. Separating the definition of the parallel behaviour from the source code allows the I/O and computational behaviours to be defined in an abstract manner. This definition allows a global optimization of all parallel behaviours of an application rather than a local optimization of a specific parallel behaviour.

Because of the interdependence between the parallel behaviours of I/O and computation, I/O templates will need external information about the computational parallelism and the run-time environment to efficiently perform their task. For example, the definition of an I/O block is important. Other elements of information needed (and why) are: which processes are participating in the file access (coherency and synchronization), what function each process will perform (program ordering), and what work remains to be done (prefetching and clustering). Architectural information such as locations of physical file systems, processor types, and other run-time information (like network and processor loading) is also required. The I/O templates are intended to be integrated into the parallel programming system. The PPS collects and manages this information for the templates at both compile-time and run-time.

Section 3.1 describes the proposed parallel I/O templates. Section 3.2 presents the read and write attributes of the model that are used to enhance performance and tune the model for a given application. Section 3.3 uses an example to show how templates can be used to compose more complex parallel I/O access patterns. Section 3.4 discusses what external parallel computation information is needed for these I/O templates. Section 3.5 summarizes this chapter.

3.1 Description of Parallel I/O Templates

The model currently contains five parallel I/O templates. The tree (Figure 3-1) is a class hierarchy diagram for these templates. The templates are similar to Crockett's proposal of independent, segmented, and global file I/O [23]. **Sequential I/O** class behaviour is the root of the tree. Sequential I/O has no parallel behaviour. With **Independent I/O** behaviour, each participating process has its own file pointer that it can move independently. There is the potential for each process to synchronize at the beginning (starting point) and at the end using the file pointer if the file is modified. With **Segmented I/O** each of these independent file pointers is restricted to its own file segment. With **Global I/O**, each of the sequential file pointers is synchronized with a single global file pointer. Each of the shaded parallel templates in Figure 3-1 adds synchronization constraints to these basic abstract parallel behaviours.

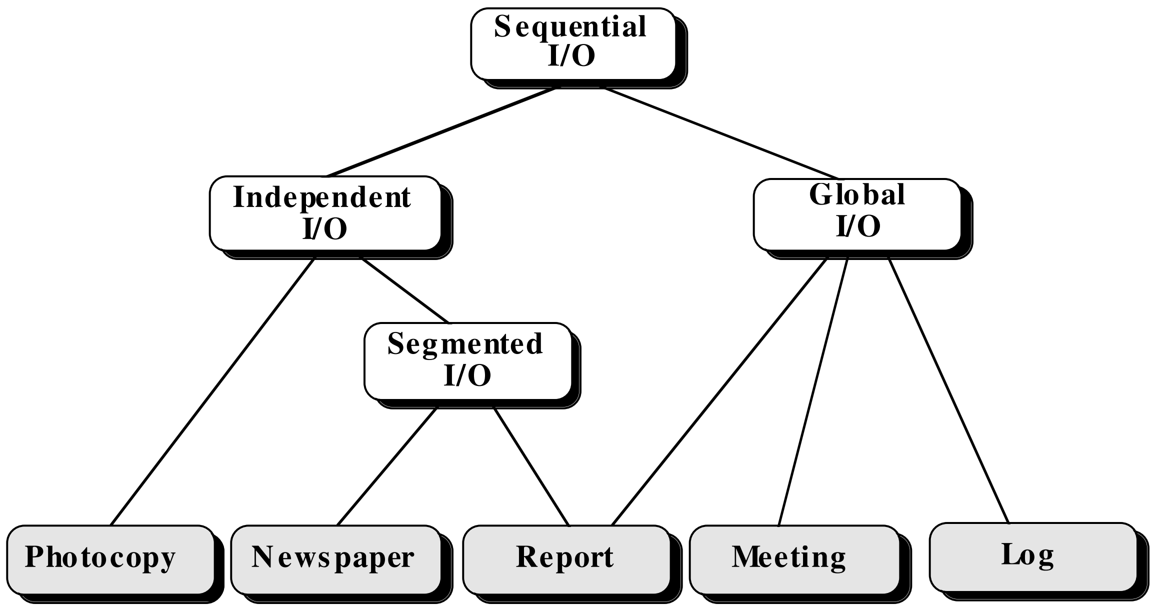

Figure 3-1 — Parallel I/O behaviour hierarchy.

This collection of templates is not complete. As applications using the templates are developed, the set can be extended if new behaviours are needed. However, it has not proved necessary to add new behaviours to the set at this point. One reason that new templates have not been necessary is that the current simple behaviours can be composed together to represent more complex I/O access patterns.

The next five sections briefly describe the proposed templates. Each section contains a description of the parallel behaviour and any special characteristics of the parallel behaviours. It is important that users can easily understand the semantics of each template. Analogies are provided to illustrate the meanings of the templates. The templates are intended to be integrated with parallel computational templates to create a parallel application.

3.1.1 Photocopy

A familiar situation arises when an author distributes copies of a paper for review. After the reviewers have made their comments on their private copies, the author integrates all or some of the changes back into the original document. This may take several iterations. A **photocopy** template is intended for independent file access.

Knowing that a file is a **photocopy** opens up the possibilities for exploiting this information to some I/O optimizations. For example, since multiple processes read the same file, one optimization is to selectively replicate the file so that the I/O operations become local I/O instead of networked I/O. However, a **photocopy** has the property that any write operation must be verified by the owner or controller of the file before becoming visible to any other processes.

3.1.2 Newspaper

A newspaper is composed of sections that can be read (or written) independently. The **newspaper** template is a way to segment a file into independent pieces with no overlap. Each process gets exclusive access to a portion of the file. Any process that reaches the end of the segment has reached its version of the end-of-file. It is an error to exceed the segment boundaries. A **newspaper** file segment always knows its starting point (**base**) in the file and size of the segment (**extent**).

The **extent** is determined at run-time by a user-supplied function or constant value. In the special case that the size cannot be determined in advance of using the segment, the segment is said to be **unknown**. **Unknown** segments are denoted by a size of zero. The file manager determines what, if anything, will need updating after a process is finished using an **unknown** segment. If an **unknown** segment is modified, the file manager appends it to the file, based on some user-defined ordering attribute. The **base** of the segment is not used after the first append. However, if the **unknown** file segment is unmodified, the file manager advances the file pointer to the current position returned by this **unknown** segment based on a user-defined ordering attribute.

The drawback to an **unknown** segment is that any process not in its group will block at the next I/O statement that uses this file pointer until all the outstanding I/O by the group is received by its file manager. However, if the system can determine that a given I/O statement does not access the **unknown** section of the file, it does not need to block. Although **unknown** segments require more complicated synchronization, they are supported because of the variable length C stream I/O operations. It is only after the **printf** or **scanf** completes that the user can determine the new file position.

3.1.3 Report

Having a group write a report usually involves the members collectively reading several other sections prior, during, or after writing their own sections. As well, comments may be written to other sections of the report which may or may not be incorporated into these remote sections. A **report** template has both global and segmented file properties. The global property is that only one process is allowed access to a particular file segment at a time but there is a mechanism for sharing access. The segmented property means that the file can be divided into "independent" segments. However, two processes can exchange segment ownership if necessary. That is, no segment has a fixed owner. A process must obtain read or write permissions for the desired segment from a **file manager** (parent). This behaviour provides a protocol so that multiple processes are able to concurrently access overlapping regions of a file.

With this template, if a process exceeds its local segment boundary limits, it is not immediately considered an error. Rather, it is a signal to the PPS that one segment is to be exchanged for another. An error that is reported back to the user occurs if the file pointer is moved to point before the **base** of the first file segment or past the **extent** of the last segment. If the PPS determines that exceeding the limits is not to be an error, the process asks the manager of the **report** file pointer for the appropriate permissions (read or write) to access the file at the new location. There is a hierarchical structure so that if a given manager cannot resolve the permissions because they are outside of the current manager's boundaries, the manager will ask its file manager to resolve the request.

Like the **newspaper** template, the **extent** of a given segment is determined at run-time by a user-supplied function or constant. Similarly, an **unknown** segment size is resolved after the process returns the segment to the process's file manager. However, unlike the **newspaper** template, modified segments are not appended to each other. The segments are overlaid based on a user-specified ordering attribute. All processes not participating in this **unknown report** group are restricted from using the **report** file pointer until the entire group is finished and the file state is resolved. Again, if the system can determine safe access to portions of the file, this restriction can be relaxed.

3.1.4 Meeting

The analogy comes from a meeting where only one person has control of the floor at a time. The **meeting** template uses a global file pointer and all processes using it must synchronize and coordinate access to the file. A **meeting** has both global read and write capabilities. However, only the process that has control of the file may read or write at any time.

If more than one global file pointer is passed to a process (e.g. an input and an output file pointer with global semantics), there may be a problem of deadlock. Therefore, to prevent deadlock, when a process asks for control of one global file pointer, the process must receive control of all the other global file pointers involved in the transaction. For example, suppose two processes, $\mathbf{P_1}$ and $\mathbf{P_2}$ are sharing two file pointers, $\mathbf{f_1}$ and $\mathbf{f_2}$. $\mathbf{P_1}$ asks for access to $\mathbf{f_1}$ and gets control of $\mathbf{f_1}$. Because of program flow, $\mathbf{P_2}$ first asks for access to $\mathbf{f_2}$ and receives control of $\mathbf{f_2}$. Now, if $\mathbf{P_1}$ asks for control of $\mathbf{f_2}$ or $\mathbf{P_2}$ asks for control of $\mathbf{f_1}$, deadlock occurs unless either process first releases the file pointers that the process currently controls. This release may not always be immediately possible.

3.1.5 Log

The analogy is with maintaining a record of events. Once an event has been recorded, it is never modified. The **log** template uses a global file pointer with the added restriction that all write activity takes place at the end of the file. After a write takes place, the file pointer is left at the end of the file. However, any read or seek operation is free to proceed without synchronization because the data is always consistent. The global end-of-file (EOF) marker is moved only when the process with the current access permission updates the global data structure. All other processes are limited to the last known value of the global EOF.

3.2 Read and Write Attributes

In addition to a template's base semantics, each template can have several attributes which refine its behaviour. One attribute of all the parallel I/O templates presented here is the ordering of I/O operations. Read and write operations can have separately defined ordering. That is, the order in which a collection of processes communicate with each other defines both the access sequence and when updates become visible. There are three possibilities: **ordered**, **relaxed**, and **chaotic**. These correspond to the three levels of program correctness for I/O— sequential, serial, and chaotic — defined in Chapter 2.

For example, the source code in Figure 3-2 performs blocks of I/O in the order: α_0, α_1, α_2, β_0, β_1, β_2, using the two loops. Both `Alpha` and `Beta` are remotely executed functions. For the moment, assume that there are six separate processes that are concurrently executing three copies of `Alpha` and three copies of `Beta`. For convenience, each process is identified as: α_0, α_1, α_2, β_0, β_1, β_2.

The **ordered** I/O attribute means the I/O using this file pointer will be done in sequential order. If `fp` is a global file pointer, the ordering attribute will sequentialize the I/O. If `fp` is an independent file pointer, all changes to the master file are recorded in program order. That is, a write by α_0 is seen by subsequent file accesses by the other processes. However, changes to the file by α_1 are not seen by α_0, even if α_1 finishes before α_0. At the end of the computations, the file contains only the output of β_2 as all the processes start at the same location in the file and only the last modification will remain. If `fp` is a segmented file pointer, all six pieces of work will be sent out to execute concurrently as independence is ensured by enforcement of the boundary conditions of each segment. In this case, the file will look like α_0, α_1, α_2, β_0, β_1, β_2 regardless of the order in which the individual processes finish. If the length of each segment is **unknown** at the time of the remote function invocation, the **ordered** attribute ensures that the modified segments are appended to the file in sequential order.

If the template has both global and segmented characteristics (a **report**), the **ordered** attribute ensures that the requests for access to other segments are processed in sequential order. That is, the process accessing segment α_2 that now needs to access the segment α_0 must first surrender segment α_2 to the file manager. Then, the process that just surrendered α_2 waits until the process doing α_1 is finished or indicates it needs no further access to

```
AlphaBeta ( FILE *fp )
{
    int j ;
    for ( j = 0 ; j < 3 ; j++ ) {
      Alpha ( fp ) ;                  /* The Alpha I/O done in parallel */
    }
    for ( j = 0 ; j < 3 ; j++ ) {
      Beta ( fp ) ;                   /* The Beta I/O done in parallel */
    }
    fclose( fp ) ;            /* Close the file after all the work is done */
}
```

Figure 3-2 — Sample code for I/O attributes.

segment α_0. If **unknown** segments are used, the file manager updates the master file based on program order with the end result of the file containing β_2. Read access to the shared segment is not normally blocked. However, if a process modifies the segment, the modifications are not immediately visible until the master file processes the modified segments. During this processing, all processes are denied access to this segment. For modified segments, the segments are merged instead of appended. This reflects the global nature of the **report** template. The position of the last segment file pointer sent out is used for the new master file pointer. If the segment was read only, the position of the master file pointer is updated based on the return status of the last segment sent out.

Using the **relaxed** I/O attribute, the ordering is eased somewhat. Now, the α_i I/O operations are serialized followed by the serialized β_i I/O operations. If `fp` is a global file pointer, all the α_i I/O blocks must be finished before any β_i I/O block is allowed to proceed. The α_i I/O blocks can occur in any order and will be followed by the β_i I/O blocks in any order. For example, one possible result is that the file is accessed in the following order: α_0, α_2, α_1, β_2, β_0, β_1. An independent file pointer will see any α_i changes as they are submitted to the master file controller but β_i I/O operations will see the changes only after all the α_i changes have been recorded in the master file. This does not mean the processes doing the β_i blocks are blocked. Rather, these processes proceed using the older version of the file data. This means, as well, that changes to the file by any β_i must wait until after all the α_i changes have been recorded. For example, depending on the order in which the various processes updated their contents to the master file controller, the file could be left containing the output of β_1 when the application exits instead of β_2, as seen using the **ordered** attribute.

Pure segmented file pointers will see little difference between the **ordered** and **relaxed** attribute unless the segment length is **unknown** at the time of the remote function invocation. In this case, the **relaxed** attribute requires that the modified segments are appended as-received, subject to all α_i segments being appended, followed by the appending of the β_i segments. That is, the file again could be left in the state: α_0, α_2, α_1, β_2, β_0, β_1. A **report** file pointer with an **unknown** segment length merges the α segments on an as-received basis and will merge the β segments only after the α blocks are all merged. For example, using the above ordering, the file would be left in the state β_1.

For **report** file pointers with a known segment size, any process accessing the α_i segment which then determines it needs to access segment α_j can proceed to request access for the α_j segment. Permission is granted based on a first-in-first-out ordering amongst the α processes. However, any β_i process that needs to access an α_j segment must block until all α segments have been released before proceeding. The run-time system has to determine when all α processes are finished before the request by a β process type can be granted. However, the file will be left in the same state as the pure segmented file pointer.

With **chaotic** I/O, the ordering is completely relaxed so that any process can have access to the file at any time, subject to the parallel behaviour. The global template still means that only one process has exclusive access at a time. However, program order is ignored at run-time. For example, the file could be accessed in the following order: α_0, β_2, α_2, β_0, β_1, α_1. The independent file pointer allows any update to the master file to be immediately visible to all other processes sharing this file. In this case, the updated file is left containing α_1, which was the last process to update the file.

Pure segmented file pointers with a positive non-zero **extent** are not affected by **relaxed** ordering as the segments are determined by the program ordering and are consumed in that order. The file will remain in the same state as the **ordered** I/O attribute: α_0, α_1, α_2, β_0, β_1, β_2. The request for access to another segment by a **report** file pointer is handled on a first-in-first-out basis. However, there is no blocking based on process type (e.g. `Alpha` and `Beta`). Any pure segmented file pointers with **unknown** length segments are merged in as-received order, regardless of type. For example, the file could be accessed in the following fashion: α_0, β_1, α_1, α_2, β_0, β_2. For **report** file pointers, the last segment received indicates the size of the segment used in subsequent I/O operations. In this case, the file contents were left in the state β_2, which was the last update of the file.

The ordering attribute does not modify the base behaviour unless some synchronization was required in the template. The ordering attribute does affect the way a file is accessed and modified. Depending on the type of parallelism chosen, a process may or may not have to give up access or wait for access to a given file descriptor.

3.3 Composing Templates

One of the benefits of using templates lies in the fact that they can be arbitrarily composed to support more complex I/O behaviours. Figure 3-3 shows an example which benefits from composition. In this example, a file is segmented so that concurrent processes access different portions of the file. A pipeline model of computation could require such an access pattern.

The file is divided into three segments using a **newspaper** template. Within a given segment, a portion of that segment is independently read by several other processes. Each segment is treated as a **meeting** (global file) until a particular portion of the segment is reached. At that point, several processes are granted independent access as **photocopies**. After the independent operations are finished, the file access reverts back to a **meeting** (the global file pointer forms a barrier).

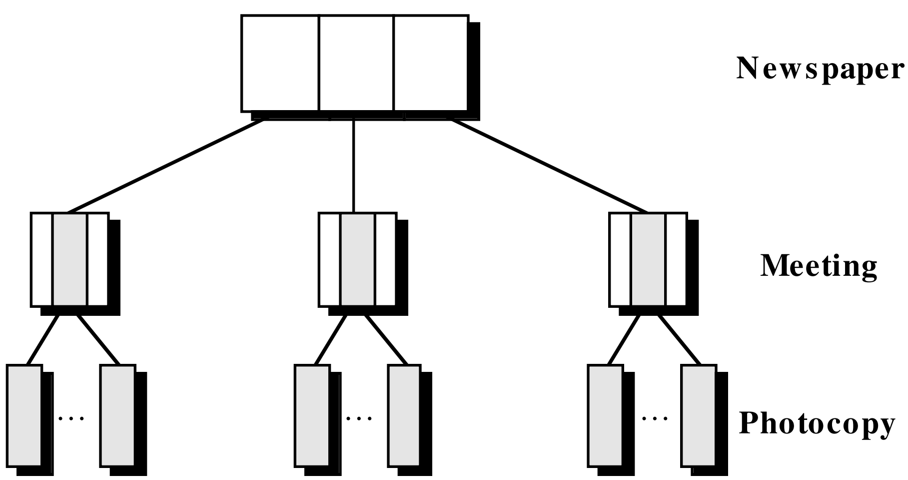

Figure 3-3 — Composing with PI/OT.

If the user tries to code all of this by hand, the amount of specialized code increases at each level, along with the chances of introducing errors. If the computational parallelism changes, the restructuring of the code to reflect the changes is a potential source of errors. For example, suppose this pipeline has sufficient granularity to run efficiently on a shared-memory multiprocessor system. If the code is then ported to run on a network of workstations, the coarser granularity required to run efficiently may mean that one stage of the pipeline should be collapsed. The **photocopy** template, along with the parallel computational behaviour of that associated stage of the pipeline, could be dropped. The system should be able to compensate for the loss of parallel I/O behaviour at that stage by integrating the code into the earlier or later portion of the pipeline. The strength of a template approach for both the computational and I/O parallelism is that any changes, either by the user or by the PPS, are quickly and correctly implemented.

3.4 External Information

The I/O templates presented in this dissertation do not exist in a vacuum. They are intended to exist and cooperate with the computational parallelism and the physical system the application will be running on, using the PPS as the management tool. External information, either explicitly supplied by the user or implicitly acquired by the parallel programming system, is needed to efficiently implement the desired I/O behaviours. The model must be able to withstand contact with the real world.

The external information can be divided into two parts: the semantic content of the program (program flow) and the physical domain used to execute the parallel application (network, file systems, and processors). Both have static and dynamic components.

The first piece of semantic information required is the definition of an I/O transaction. A transaction is two or more I/O statements that must be considered as one I/O block. I/O rarely occurs as a single operation; instead, several I/O operations are clustered as one block. For example, a seek is followed by a read or write operation. Although they are two sequential I/O statements, this is often considered as one I/O operation in the parallel domain (some systems provide an atomic seek–and–read function or equivalent to solve this e.g. [53, 57, 61]). As well, a variable length record is often read or written in two or more steps. The first operation determines the number of elements; subsequent operations read or write the elements. Alternatively, the list of elements is read or written until terminated by a special end-of-record character. Particularly in the parallel domain where concurrent processes will share access to a common resource, the definition of I/O transactions is critical.

The user can recode the application so that individual I/O steps use a temporary file or memory buffer. The parallel I/O is then done as a single operation per process. However, this does not remove the necessity of identifying the I/O blocks, since it makes the programmer identify a transaction and explicitly provide the synchronization.

Once the I/O transactions are identified, the second piece of semantic information is the program flow, as it pertains to parallel I/O. The necessary information includes both the temporal and collective constraints. From the source code found in Figure 3-2, the read and write attributes will benefit from knowledge such as which of the α I/O blocks must be finished before any β I/O block can start. As a second example, realizing that all the I/O for a matrix will be done at the same time by a group of processes allows the PPS to optimize both fetch and merge operations.

Program flow can be determined at both compile-time and run-time. Prefetching decisions are easier to schedule if the program flow is known. Static program analysis allows the system to determine the program ordering for different classes of processes. Consider the case of a computational pipeline consisting of three process types, Π_1, Π_2, and Π_3 (Figure 3-4a) that share a file. At run-time, there can be one or more instances created of each process type. Figure 3-4 shows some possible connection configurations.

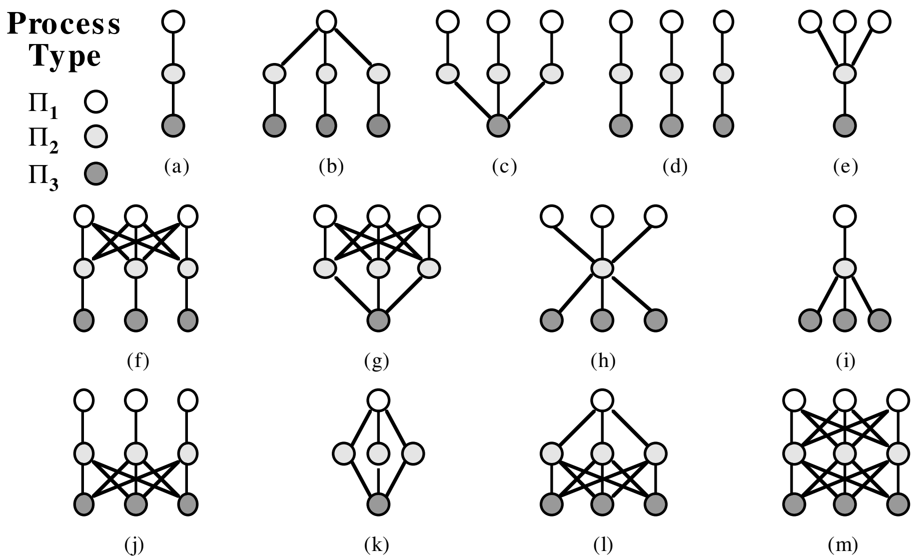

Figure 3-4 — Examples of connection patterns for a pipeline of three process types.

Each instance of the process types shares or manages its I/O information with one or more of the instances of any of the process types. Depending on the relationship between process instances (determined by the connection patterns), either a particular instance is a recipient or a maintainer of the information. To make good decisions and efficiently manage the I/O information, the configuration of the computational parallelism is needed. In addition there are three questions that must be addressed. How are the different process types related to one another? How are the instances connected? How do the parallel I/O requirements affect this connection pattern?

In the connection pattern shown in Figure 3-4d, there are three independent pipelines sharing access to a single file. If a segmented or global parallel I/O behaviour is defined for the shared file, the computational connection pattern is insufficient for the I/O parallelism as the Π_1 processes must now coordinate I/O access. Figure 3-5 shows the new connections that must be made to all the Π_1 processes to ensure synchronization and coordination of file access. Depending on the implementation, a new manager process may be needed too.

In the connection patterns shown in Figure 3-4 for a simple pipeline computational model, an instance of process type Π_1 may need to share global I/O information with the other Π_1 instances. This same instance will need to distribute I/O information to a group of Π_2 instances, and will need to synchronize the returned I/O information from this group. A group can contain one, some, or all of the various process type instances. An instance of a Π_2 process receives I/O information from a Π_1 process and eventually returns to it the updated I/O information. This Π_2 instance also distributes I/O information to a set of Π_3 instances. The Π_3 instances receive I/O information from a specific Π_2 process and will return the modified I/O information. Relying on static information is insufficient since it is possible that the actual number of instances for each process type are determinable only at

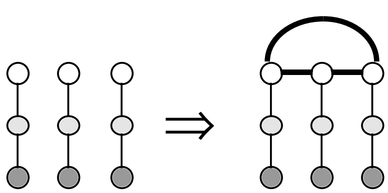

Figure 3-5 — Additional I/O communication connections needed for synchronization and coordination of file access for global or segmented parallel I/O.

run-time. It is also possible that the connection pattern can change at run-time. This could happen if a process is able to run more than one process type, depending on directives from the PPS.

The types and number of process interconnections are important since these determine how the file is to be shared. For example, in the case of a two stage pipeline, there are eight ways of connecting the process instances that are of concern to parallel I/O. They are: one-to-one, many-to-one, one-to-many, many-to-many, and replication of the previous four cases. This last step takes one of the previous four patterns and creates **n** replicas. The data file is now shared by not just two process types, Π_1 and Π_2, but by $\mathbf{n(i+j)}$ processes where **i** is the number of Π_1 processes and **j** is the number of Π_2 processes.

When a pipeline is increased by one to three process types, the number of possible combinations grows to thirty two (Figure 3-6). The boxed pairs indicate that the contents of the box are replicated as one unit. Figure 3-4 clarifies the connection patterns for some of the combinations given in Figure 3-6. The shaded areas in Figure 3-6 correspond to the figures in a left to right fashion and down each block to each configuration in Figure 3-4. For example, using a value of three for both **n** and the replication factor of the boxes, the entry in the second block, first row, and third column is represented by Figure 3-4j while the first block, second row, second column in Figure 3-6 corresponds to Figure 3-4f.

It is clear that handling all cases by hand is a difficult task. Even with a simple programming model, the growth of possibilities is exponential. However, breaking down the problem into simple pieces and having a simple set of rules to create more complex I/O models is an approach that works well for source-to-source compilers. At run-time, these rules are useful when the application performance is not predictable or irregular.

Dynamic work allocation or process scheduling can compensate for an irregular load distribution. A slower or heavier loaded processor could do **n** blocks of work in a given period of time while a faster or lightly loaded processor could do greater than **n** blocks. Alternatively, using homogeneous processors, the work load per I/O block may not be constant. Since this performance information is not always available at compile-time, run-time information is beneficial in making more intelligent prefetching, segmenting, or synchronization decisions.

The I/O templates need to determine the number of distinct concurrent processes that will collectively share access to a particular file. This grouping may contain subgroups of processes and the membership can be dynamic. For example, global or segmented I/O par-

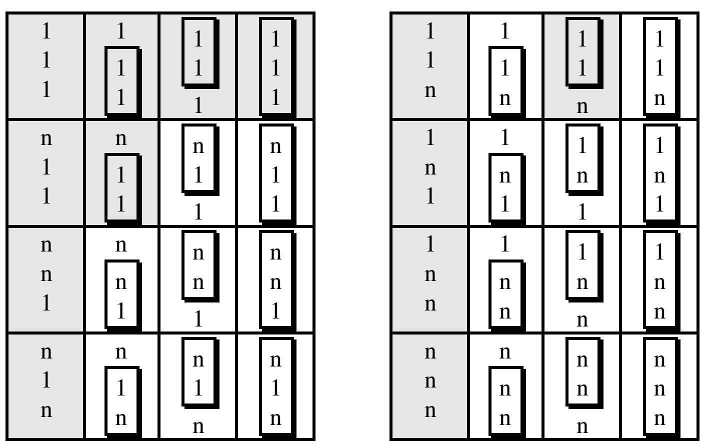

Figure 3-6 — Possible connection patterns using a pipeline of three process types. The character **1** (one) indicates a single instance of a process type while the character **n** indicates more than one process instance. The boxed pairs indicate that the contents of the box are replicated as a single unit.

allelism needs to know how many processes are involved so that there are effective implementations of these parallel I/O behaviours. A global behaviour needs to know which process will get access so that it can update all the others; a segmented behaviour may divide the file based on the number of processes or tune the prefetching algorithm; an independent behaviour can use the number and location of processes to determine if replication and local caching of the data file would be more efficient.

Physical information is not only important for determining the performance of a given template, it is also critical for computational performance. In the sequential domain, I/O is often buffered for efficient operation so that the expensive physical I/O operations are deferred until necessary. This efficiency is actually a hindrance when concurrency is added. In order to avoid overlap, a process ($\mathbf{P}_2$) needs to realize the scope of another process's ($\mathbf{P}_1$) I/O operation before performing $\mathbf{P}_2$'s I/O. The only contact between the two processes is the physical file through the two local file pointers. Deferring the physical update for the sake of the efficiency of the $\mathbf{P}_1$ process can detrimentally affect the overall efficiency of the application since $\mathbf{P}_2$ may have to wait. Of course, the $\mathbf{P}_1$ process can use deferred output but the operating system cannot be relied upon to update the physical file in a time-critical manner for other processes. Rather, the PPS must ensure that the physical update is done when accurate information is needed by the other processes sharing this resource.

Figure 3-7 gives an overall view of the PI/OT model and demonstrates how it fits into a parallel programming system. The parallel compiler modifies a sequential file pointer that has been given parallel characteristics and creates a parallel file pointer. The parallel compiler, using the parallel I/O and computational specifications, creates a parallel application. When the user runs the parallel application, the dynamic information is collected by the run-time system; using the parallel specifications, the run-time system coordinates access between the parallel application and the physical files.

Both static and dynamic analysis will benefit from physical information such as: the amount of local disk space available for caching; the physical location of the file systems on

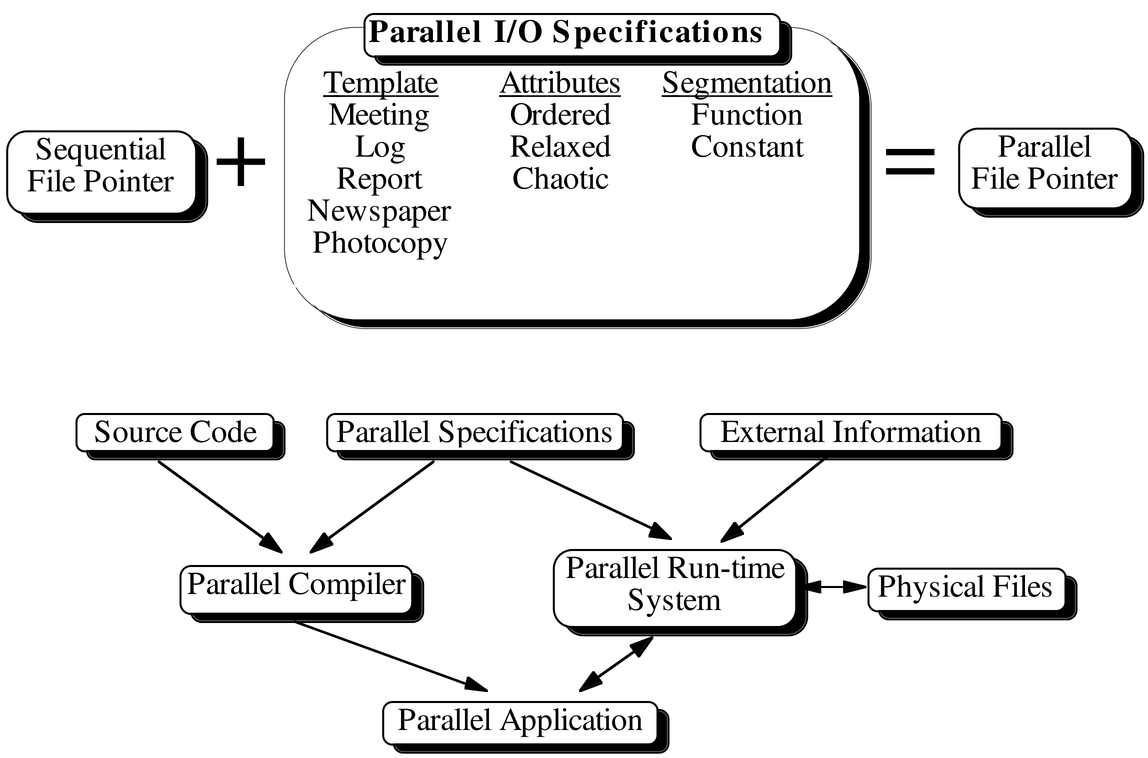

Figure 3-7 — Overview of the PI/OT model in a parallel programming system.

the network in relation to the compute processes; the type of file system — general or specialized; the network bandwidth available for I/O either to minimize the impact of the application on other users or to maximize the performance of the application; the heterogeneity of the processors — architecture and speed; and the distribution of the processors over the network. Whether these physical details can be compensated for or effectively utilized depends on the implementation. However, they will all have an impact on the overall performance of the application.

3.5 Chapter Summary

Five parallel I/O templates have been presented, each encapsulating a simple parallel behaviour. An analogy is provided for each template to make it easier to understand. Read and write ordering attributes permit tuning of the application without changing the original source code. Complex I/O patterns can be built by composing the simple behaviour of the templates. The computational parallelism and I/O parallelism can be combined to create arbitrarily complex access patterns that are still modifiable without requiring changes to the user's source code.

External information about the application and run-time environment is necessary to ensure correct I/O behaviours and performance optimizations. Compile-time analysis of the user's application (using the supplied parallel I/O specifications) inserts hints to the run-time system about the needed parallel I/O behaviours and the start and finish of any I/O transactions. At run-time, the PPS uses these hints and, based the program's dynamic environment and computational behaviours, implements the parallel I/O requirements. By requiring the PPS to either supply or determine the necessary external information, a user has minimal impact on **how** the parallel behaviours are implemented. A user selects **what** parallel behaviour is required. As well, the PPS is able to tune the performance of the entire application based on all of the user-supplied requirements and the run-time environment.

Chapter 4 explains **how** the parallel I/O templates are intended to be implemented in general and explicitly deals with one particular parallel programming system.

Chapter 4

4. Implementation

An implementation is needed to validate and to demonstrate the usefulness of the parallel I/O model presented in Chapter 3. To be considered successful (other than providing the correct behaviour), the implementation should give reasonable performance for minimal effort on the part of the user and ensure that changes to the parallel I/O behaviour be as simple as changing the template. The user should not need to differentiate at the code level between sequential and parallel I/O calls.

The parallel I/O model proposed in this work requires information regarding the parallel computations and the state of the underlying communication libraries. There are static and dynamic components to the implementation. The static or compile-time portion consists of defining the parallel behaviour and integrating it with the corresponding static component of the parallel computational behaviours. The dynamic component of parallel I/O consists of the process states of the parallel application. This includes both the relationships and the dependencies between computational processes as well as the process-processor mappings. This information, managed by the run-time component of the parallel programming system (PPS), determines the degree of parallelism and the dynamic behaviours of the running application.

There are two basic premises for implementing PI/OT. The first is that a user does not identify any parallel I/O in the source code. The parallel specifications are stored separate from the code. Second, PI/OT does not roll back I/O operations or computations. Deadlock prevention rather than deadlock detection is the approach used when implementing PI/OT.

This chapter describes the implementation of PI/OT. It is divided into three main parts. The minimal PI/OT requirements are presented in Section 4.1. These are the minimal system (user-interface, compile, and run-time) requirements for any general PPS implementing PI/OT. Section 4.2 describes how the PI/OT programming model is intended to be implemented from an I/O functionality viewpoint. Section 4.3 describes the PI/OT model implementation from a template viewpoint. Section 4.4 describes the specific modifications made to the Enterprise PPS (user-interface, compile, and run-time) for PI/OT. Section 4.5 describes the current deadlock prevention mechanism. Section 4.6 gives a summary of this chapter.

4.1 PI/OT Minimal Requirements

There are a number of conditions that must be met prior to implementing PI/OT in a given parallel programming system. The primary requirement is the ability to send, poll for, and receive asynchronous messages of arbitrary size between distinct processes in a reliable manner. The distinct processes requirement does not necessarily imply traditional heavy-weight UNIX processes; a thread would qualify as well. Inter-process messages are needed for synchronization and coordination of a data file with the parallel application. The application views a file as a single global entity regardless how it is physically stored.

The PPS run-time system is also responsible for providing unique process identifiers so that PI/OT can use the communication system to send a message to, poll for a message from, or receive a message from any specified or arbitrary process in the parallel application without blocking. This implies that the user or the PPS has divided the application up into concurrent computational tasks.

The PI/OT implementation has to be able to intercept and substitute data in messages for these computational tasks that contain file pointers. Modifications are made both at the time of sending and at the time of receiving the messages in order to insert or extract the PI/OT information. Substitutions are made in addition to the other PI/OT messages necessary to implement the various parallel I/O behaviours. PI/OT does not force the user to learn a separate set of parallel I/O functions. Rather, the user writes code using the familiar UNIX `stdio` interface (or low-level I/O interface). A source-to-source translator (**precompiler**) identifies the user-specified parallel I/O file handles. The precompiler needs to create the necessary hooks to integrate any user-supplied segmentation functions into the PI/OT run-time environment. The PI/OT run-time system is responsible for sending, processing, and returning the dynamic state of an I/O object between cooperating processes.

All tasks in the parallel application are classified by both their computational and parallel I/O functionality. The PPS run-time system is responsible for mapping tasks and files to processes and processors. It must also support queries to specific processes or tasks by the PI/OT implementation. For PI/OT, a user identifies the parallel file descriptors and their parallel behaviour for each task classification. For example, a given file descriptor is designated as having global I/O semantics for a particular task. Another class of tasks designates a file descriptor with segmented I/O semantics. At run-time, the two file descriptors are joined by the first task passing the global file descriptor to the second and linking the two together (a remote function invocation). The PI/OT run-time system must resolve the apparently different behaviours.

It remains the responsibility of the user to define **what** the computational and I/O specifications are, and leave the precompiler and PI/OT component of the PPS run-time system to define **how** to implement them.

In summary, PI/OT requires a message passing system that can poll, send, and receive messages of arbitrary length in a reliable fashion. Parallel tasks must be identified and mapped to processes in some fashion available to PI/OT. The parallel I/O is identified on a per-task basis. Any messages sending a file-descriptor must be identified and have the I/O component replaced with the parallel I/O configuration for that component. Similarly, a message containing a parallel I/O data structure must be converted into a conventional sequential file pointer for the user's code. This is best done using a mixture of compiler and run-time support. Finally, the PPS run-time system must be able to spawn tasks (either as a thread or heavy-weight process) when necessary.

4.2 PI/OT Implementation Issues

This section describes how the parallel I/O templates are intended to be implemented. Users of the templates do not need to know about a given implementation and alternate implementations can be used without affecting user programs. The PI/OT templates are intended to parallelize the standard C stream I/O library. Figure 4-1 gives a listing of the various standard stream functions and their signatures. However, there is no reason why the templates could not be implemented to replace low-level I/O calls (`open`, `close`, `write`, `read`, `lseek`).

Each process maintains a list of the local parallel file pointers it uses. This list is called the **ParIO** list. Each list element contains information such as the template type, the current file state, whether access is permitted, the communication handle of the local process and the manager process.

A run-time list of active and outstanding outgoing parallel I/O requests is maintained, either on a per-process basis or on a per-thread basis if the process is multi-threaded. This list is referred to as the **call-chain**. Each list element contains information such as the parallel I/O template, the current file state, whether access is permitted, and the communication handle of the caller and callee. There is a corresponding list, called the **pending** list,

```
#include <stdio.h>
/* Opening streams */
FILE *fopen( const char *filename, const char *type ) ;
FILE *freopen( const char *filename, const char *type, FILE *stream ) ;
FILE *fdopen( int filedescriptor, const char *type ) ;
/* Closing streams */
int fclose( FILE *stream ) ;
/* Flushing streams to or from disk */
int fflush( FILE *stream ) ;
/* Moving the file pointer in a stream */
int fseek( FILE *stream, long offset, int ptrname ) ;
void rewind( FILE *stream ) ;
long ftell( FILE *stream ) ;
/* Testing file stream for end-of-file */
int feof( FILE *stream ) ;
/* Reading from a stream */
int scanf( const char *format, ... ) ;
int fscanf( FILE *stream, const char *format, ... ) ;
size_t fread( void *ptr, size_t size, size_t nitems, FILE *stream ) ;
/* Writing to a stream */
int printf( const char *format, ... ) ;
int fprintf( FILE *stream, const char *format, ... ) ;
size_t fwrite( const void *ptr, size_t size, size_t nitems, FILE *stream ) ;
/* Get a character, word, or variable length character string from a stream */
int getc( FILE *stream ) ;
int getchar( void ) ;
int fgetc( FILE *stream ) ;
int getw( FILE *stream ) ;
char *gets( char *s ) ;
char *fgets( char *s, int n, FILE *stream ) ;
/* Put a character back into the stream */
int ungetc( int c, FILE *stream ) ;
/* Put a character, word, or variable length character string to a stream */
int putc( int c, FILE *stream ) ;
int putchar( int c ) ;
int fputc( int c, FILE *stream ) ;
int putw( int w, FILE *stream ) ;
int puts( const char *s ) ;
int fputs( const char *s, FILE *stream ) ;
```

Figure 4-1 — Standard C stream I/O library function signatures.

which is maintained for I/O messages from other processes. For example, a process may require access to a global file or it may be returning access permissions for a given file segment. The receiving process may not be able to process the message right away and it is queued in **pending** for later processing. Another process has the access permission and will return it at some point in the future. When an access permission message is received, the **pending** list will be processed for any requests that can be satisfied.

It is important for the **call-chain** and **pending** lists to be consumed in an order that preserves program correctness. If a task does not consume the **call-chain** list, the outstanding calls will need to be cancelled. (This happens if there is an error condition that ends the task's work or a task is generating speculative work and any answer is sufficient to end the task.) Similarly, any **pending** requests will need to be cancelled and the calling processes will have to take any necessary actions to recover. (Normally, the I/O function returns a failure.) Note that the computational component of the PPS will need to end any outstanding computational tasks at this time as well.

The parallel file pointers are not explicitly differentiated from the sequential ones in the source code. What is the minimal amount of information necessary to determine if a file variable has parallel behaviours? All I/O functions, except those of the **open** variety, require either an explicit or implicit file handle passed as one of the parameters (for example, `fprintf` or `printf`). Since all I/O handles are associated with a unique number that is assigned by the operating system, looking up the file number in a list can obtain the indicated parallel behaviour. This implies that multiple parallel behaviours are not permitted on a single file descriptor. However, multiple file descriptors (and thus parallel behaviours) can be associated with a single file.

Inter-process communication uses this unique file descriptor to match a parallel file descriptor from a remote process with a local file descriptor. PI/OT manages two data structures which are used to match a parallel file descriptor from a remote process to a corresponding local one. The first data structure contains the local information of the parallel file descriptor. Included in this information is the unique system file descriptor which is passed to the user's code. The second data structure, stored either in the **call-chain** or **pending** list, contains the remote parallel file pointer information and the same unique system file descriptor. A message received from another process contains the remote process's parallel file information. Depending on the message type, the receiver matches the remote information in either the **call-chain** or **pending** list. The unique file descriptor is extracted and the local process parallel file structure is located.

The balance of this section is divided into five subsections. First, determining program order and instantiating the I/O managers are addressed. Second, how to grant access to a file is defined. The final three subsections discuss the run-time details of creating, closing, and using a parallel file descriptor.

4.2.1 Determining Order and I/O Managers

The I/O templates are intended to work within a hierarchy of remote message sends. Any process that makes a remote call to another process creates a hierarchy. In Figure 4-2, process **A** makes a remote call to process **B**, which in turns calls process **C**, which then calls **B** and so on. The order in which the calls are dynamically made forms the **call chain** which the I/O templates use to implement the correct parallel I/O behaviour. That is, the run-time behaviour of the application is taken into account for the parallel I/O behaviours. (The I/O managers will be defined later in this section.)

The I/O templates are implemented using a client-server model that is distinct from the computational model used. If the remote call includes a file pointer, the file pointer is treated as a parallel file pointer by the PPS. The user must specify the parallel behaviour of the passed file pointer. Another way a parallel I/O object is identified by PI/OT happens when a file is collectively opened by a group of processes. In either case, the user must identify to the PPS the intended parallel behaviour for the I/O object. Otherwise, PI/OT would either consider the call illegal or impose some sort of default behaviour.

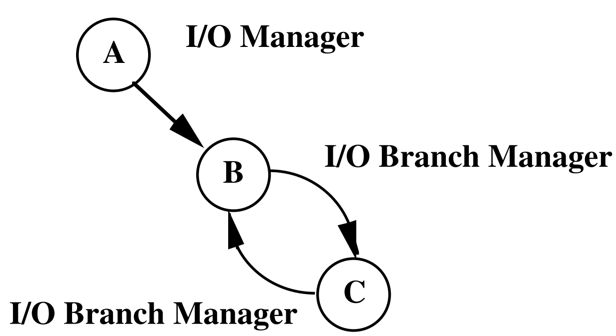

Figure 4-2 — Identifying I/O managers and call ordering.

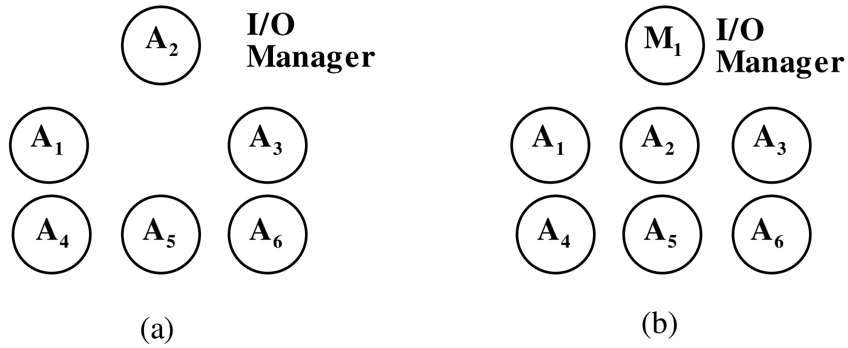

Figure 4-3 — Two approaches to selecting an I/O manager.

With improved compiler technology, PI/OT may someday be able to automatically determine the appropriate parallel behaviour. For example, if a remote procedure call is generated in a loop and contains a file pointer as a parameter, attempting a segmented I/O behaviour may be an appropriate choice if maximum concurrency is desired. This will necessitate a preliminary estimate of the segmenting factor for the remote process. Alternatively, after examining the remote function code and finding only one I/O call using the pointer, the file pointer could be designated as a shared file pointer. The remote function code could be modified to release the file back to the pool of waiting processes after completion of the I/O functions.

Every parallel I/O transaction has a **manager** (server) that is responsible for coordinating and enforcing the parallel behaviour of the group of processes (**clients**) that share the I/O object(s) composing a transaction. This happens regardless of the parallel template chosen for an individual file pointer. Normally, the process that creates and passes the I/O transaction to another process is considered the **manager**. The process that opens a file is considered the **owner**. The manager's duties range from disseminating control information to merging data. In the case of multiple or replicated processes concurrently opening the same file descriptor, the PI/OT run-time system designates one process as the I/O manager. The I/O manager process does not necessarily have to be implemented as a separate heavy-weight process. It can be collapsed as a thread into an already existing computational parallel process.

Figure 4-3 shows two approaches for selecting an I/O manager for the collective open case. (Selection of one approach is left to the implementor of PI/OT.) All of the **A** processes in the figure try to open the same file. The PI/OT run-time system decides which one of them will contain the manager task. In Figure 4-3a, $\mathbf{A_2}$ takes on the I/O manager duties because it was the first one to request the open. All the other **A** processes become clients for I/O purposes only. As an alternative, PI/OT could request that the PPS spawn a new manager process. In Figure 4-3b, the $\mathbf{M_1}$ process is created exclusively for managing the I/O for the $\mathbf{A_i}$ processes. Finally, the user could indicate to PI/OT that an I/O manager process must be placed on a specific processor (for example, if the disk file is local to a particular processor). In this case, all the **A** processes become clients for I/O purposes.

It is important to realize that a client becomes a **branch** I/O manager when the user's process (which contains the I/O client) in turn makes remote calls to other processes. In Figure 4-2, the **A** process is also the I/O manager for the call to **B**. However, when **B** passes its file pointer to **C**, **C** considers **B** as its I/O manager. Then, **C** becomes the I/O manager for the recursive call back to **B**, and so on. If information is needed, the request flows up the **call chain** until the appropriate manager can provide the information.

4.2.2 Granting Access

All template I/O operations involve several processes — one or more clients and the manager. The manager is responsible for synchronizing access and merging results. The client must recognize when access permissions are required, execute the user's code, and end the I/O transactions properly. Figure 4-4 shows some of the steps needed to exchange access permissions between two client processes and a manager.

If the file pointer is intended for sequential I/O, the I/O operation proceeds normally and control is returned to the user's code when the I/O operation finishes. If the file pointer is parallel, the client determines if it has access to the file. If it does not, the manager of the I/O object is sent a message requesting access. In Figure 4-4, both clients **A** and **B** send simultaneous requests to the I/O manager (step 1).

When the manager receives a request for access, it searches its **call-chain** list to determine if access can be granted. This list is populated in two ways. First, the manager process is informed by the computational component of the PPS that remote messages containing I/O objects are being sent. Second, collective I/O requests are received from client processes. For example, a collective open may be done using a segmented file. Each client open will receive its own segment or an error. The manager must segment the file and synchronize the merging (if necessary) of the segments after the client is finished processing.

Each entry in the **call-chain** list contains information such as the caller and callee identifiers, the parallel file pointer data structure, transaction identifier, and a time stamp. From this list, the manager can determine who collectively accesses the file pointer and in what order access is permitted at run-time. At this point, the manager determines if there is any potential for deadlock in the various transactions and prevents it from happening. Another important function of the I/O manager is to inform the caller process that it is safe to perform another I/O operation. In the example given in Chapter 1.1, all `Child` processes would share the same I/O manager. However, as `Parent` opened the file and is considered the owner of the entire file, `Parent` is not considered a client of this manager even though the manager process could be a thread in the `Parent` process. As `Parent` initiates remote I/O calls in `Child` processes, it must query the I/O manager process before executing `par_fclose`.

The access permissions for an I/O descriptor are determined by the ordering attributes of its template. If access is not allowed at this point, the request is added to the **pending** list of the manager until the request can be satisfied. If access can be granted, the manager marks the request in the **call-chain** list as **active**, updates the parallel I/O data structure, and sends a message containing the access permission along with any new global information to the client. When the client receives the manager's message, the client's file data structure is updated to reflect the new global information.

When access is granted to the client process (Figure 4-4, steps 2 and 4), the I/O operation is verified so that it will not violate any of the parallel template constraints. For example, it may not exceed the local segment's boundaries. Then, the I/O operation is executed and the parallel file data structure is locally updated.

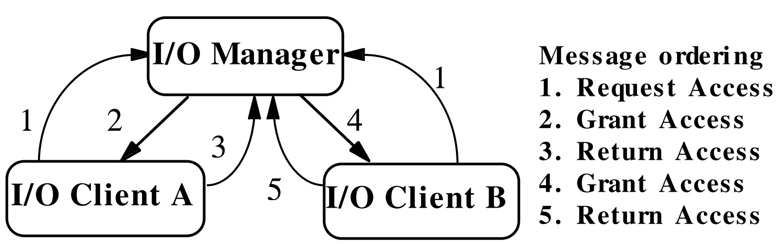

Figure 4-4 — Granting access using PI/OT.

If the atomicity of the parallel I/O operations is set to be a single I/O operation, the client must temporarily surrender control after completion of the local I/O operation. This is accomplished by sending the manager a message that the client is temporarily surrendering access. This message includes the updated global information for that parallel I/O object. The client records that its local parallel file pointer has been denied access and continues processing until the file pointer is needed again. At this point, the client petitions the manager for access permission.

If the I/O operation takes place within an identified transaction, control is retained until the transaction is finished (Figure 4-4, steps 3 and 5). The client sends a permanent surrender message and the updated parallel file pointer information to the manager. Finally, the client process returns to the user's code.

When the manager receives the access surrender message from the client, the manager searches the **call-chain** list for the active I/O object. It updates its own local parallel I/O data structure for this pointer. It then removes the I/O object from the **call-chain** list (if the surrender is permanent) or simply deactivates it. In either case, the manager then searches the **pending** list for the next I/O request that can be satisfied.

4.2.3 Creating a Parallel File Descriptor

The user's entry points into the stream I/O library are the `fopen`, `freopen`, and `fdopen` functions. Their signatures are given in Figure 4-5. After determining that one of these open requests is for a parallel stream, PI/OT must instantiate the parallel behaviour for the given file and add the resultant parallel file descriptor to its internal list (**ParIO** list) of parallel I/O objects. The typical user entry point is `fopen`. In this case, how will the PI/OT run-time system know about parallel behaviours? Since the parallel computation tasks are already identified by the PPS, any file opened by a parallel task may exhibit a parallel I/O behaviour. Adding the task identifier allows the run-time system to know how to search for parallel information.

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *type) ;
FILE *freopen(const char *filename, const char *type, FILE *stream) ;
FILE *fdopen(int fildes, const char *type) ;
```

Figure 4-5 — Three entry points for stream I/O.

A given task can have several file pointers, each with different parallel behaviours. Thus, the name of the variable assigned to the file pointer is passed to the PI/OT run-time system, ensuring that the correct parallel I/O object in the right parallel task is properly updated. Figure 4-6 shows the parallel version of `fopen` and its new signature. This signature modification can be done at compile time since both the variable name (`fpName`) and parallel task type (`parTask`) are known.

The PI/OT run-time system searches the **ParIO** list of the process to see if the variable and task name tuple are associated with a user-defined parallel I/O description. Note that this search method is different from the previously discussed way of identifying parallel file pointers by using the unique system file handle. Since the unique I/O file handle does not yet exist, an alternate way is needed to determine a unique parallel I/O description.

If the tuple can be associated with a user-defined parallel I/O behaviour, the PI/OT run-time system will determine how to best open the parallel file. What is best could range from selecting the I/O manager, to making a copy of the file local to improve performance, or to simply opening the file. The parallel open updates the **ParIO** list with the new name of the file and the file descriptor value. If the file fails to open properly, the update is not performed and a NULL file pointer is returned to the user.

```
FILE *Pilot_fopen( const char *filename, const char *type,
                   const char *fpName, const char *parTask )
{
  FILE *fp ;
  if ( Pilot_isParallel( fpName, parTask ) ) {      /* Parallel I/O */
    fp = Pilot_open( filename, type, fpName, parTask ) ;
  } else {                                          /* Sequential I/O */
    fp = fopen( filename, type ) ;
  }
  return fp ;
}
```

Figure 4-6 — Wrapper code for a parallel `fopen`.

The parallel open also takes into account whether the process is trying to collectively open the file. As pointed out in Section 4.2.1, if an open is done in a collective manner, the results are similar except that the access and access coordination are generated by the newly appointed or created I/O manager.

While `fopen` is the usual approach to opening a stream data structure, reopening a file causes the argument stream to be closed, regardless of whether it can be re-opened or not. Therefore the return stream will point to the new file or NULL. Because of the semantics of `freopen`, there are two alternatives to its parallel implementation (Figure 4-7).

Figure 4-7a shows the signature of the first implementation. This version relies on the fact that the passed file pointer has already been identified and defined for the parallel behaviour. Since a `freopen` statement is dealing with a previously opened file, only the physical file that is connected to the file pointer changes. The returned file pointer exhibits the same parallel behaviour as the passed file pointer. In this case, no modification to the signature of `freopen` is needed.

Figure 4-7b shows a second alternative which does require the signature of `freopen` to change. As the existing stream is closed regardless whether the opening of the new stream is successful or not, the new variable can be associated with a new parallel behaviour. In this case, both the parallel behaviour and the physical file can change. By passing similar information as was previously done with `fopen`, the appropriate parallel behaviour is associated with the new file descriptor and the existing file pointer (parallel or not) can be closed. If the second implementation is chosen, the function modification can be done at compile time along with the `fopen` modifications. Since this second method can also be done by a combination of the `fclose` statement followed by the `fopen` statement, the first choice of implementation should be Figure 4-7a.

The `fdopen` function associates a stream interface with a low-level file descriptor (for example, a pipe or a device). If the original file descriptor is opened, taking into account possible parallel semantics, `fdopen` will require no modifications because the look-up table has been properly updated. If the low-level open is not capable of associating a parallel behaviour with a file descriptor, the `fdopen` would be modified in a manner similar to the `freopen` and `fopen` functions.

As part of their parallel functionality, all three of these functions will update the look-up table of defined parallel I/O structures for the process (the **ParIO** list). Because the func-

```
a)  FILE *Pilot_freopen(const char *filename, const char *type, FILE *stream )
b)  FILE *Pilot_freopen(const char *filename, const char *type, FILE *stream,
                        const char *fpName, const char *parTask )
```

Figure 4-7 — Two alternative signatures for `freopen`.

tion signatures can change depending on the defined computational parallelism and the user's specifications, using compiler technology to add the additional information would be the most efficient and transparent approach.

It should be noted that this does not preclude the possibility of having multiple file pointers pointing to the same file object nor will PI/OT hinder the user from doing this. Resolution of the outcome of such behaviour is undefined. PI/OT does not coordinate behaviour between different file pointers, it manages the parallel behaviour of a single system file pointer.

4.2.4 Closing a Parallel File Descriptor

Closing a parallel file descriptor will create different actions depending on where or when the **fclose** function is called. Figure 4-8 shows the wrapper code for a parallel close function. If the file descriptor has a parallel behaviour (`Pilot_isParallel`), the run-time system tries to gain access to the file pointer (`Pilot_resolveAccess`). If the process that opens the file then attempts to close it, this function should cause the process to wait until all child processes are finished. This can be seen in the simple example given in Chapter 1.1. The `Parent` process must wait until all `Child` processes are finished before closing the file. Any I/O requests in the **call-chain** or **pending** list that involve the same file descriptor will block the parent process. The actual close is done by `Pilot_close`. The parallel close operation merges file segments into the master file (this may involve blocking depending on the ordering constraints), flushes data to disk, cleans up any temporary buffers and files, and removes the file name and file descriptor from the look-up list of parallel I/O objects in the manager process.

```
int par_fclose( FILE *stream )
{
  int status ;
  if ( Pilot_isParallel( stream ) ) {                /* Parallel I/O */
    Pilot_resolveAccess( stream ) ;
    status = Pilot_close( stream ) ;
  } else {                                           /* Sequential I/O */
    status = fclose( stream ) ;
  }
  return status ;
}
```

Figure 4-8 — Wrapper code for parallel `fclose`.

Things become more interesting if the close operation is called from a client process. After determining if the file pointer has a parallel behaviour (`Pilot_isParallel`) and resolving access permissions (`Pilot_resolveAccess`), the parallel close function is called. Recall that access is only granted to the client if the client's own **call-chain** and **pending** lists are empty of requests for the file pointer. However, for the client, the functionality of `Pilot_close` is different. The file is closed only after consultation with the manager of the parallel file pointer.

With global file descriptors (**meeting** and **log**), there is only one active process accessing the file, which removes any race conditions. Any **inactive** I/O work requests are marked as closed (i.e. the file pointer is set to the `NULL` pointer) from the manager's **call-chain** list. These are I/O requests that have not yet been assigned to a remote process. Note that the I/O requests are not removed, as would be the case if the computation ended.

The manager's **pending** list requests are processed in two ways, depending on the **order-stamp** associated with both the request and close events. (Think of an order-stamp as a variant of a time-stamp.) The events maintained by the manager's **call-chain** and

pending contain an order-stamp which corresponds to the order in which the I/O transactions were generated. For the close event, the order-stamp is the same as the one for the file descriptor assigned in the **call-chain** that was passed to the client process.

First, if the order-stamp of an entry on the **pending** list is less than that of the `fclose`, the request will be granted. That is, the work is allowed to proceed as if the file has not been closed. Second, if the close has the earlier order-stamp, the request is returned indicating that the file is closed. The client process is responsible for terminating the computation task and generating a request for new work from the manager process. It behooves the user to routinely check return codes for any I/O operation.

Independent templates (**photocopy**) inform the parent process about the close operation. The uncommitted I/O work requests using the affected file descriptor in the **call-chain** list are marked as closed. As the work is consumed, the pending requests will find the file closed.

If the file descriptor is segmented, the client will contact the manager process and inform it about the close action. All outstanding work is allowed to proceed but any uncommitted work with this file descriptor is marked as closed in the **call-chain** list. While the manager's response to the close request is proceeding, the client process closes the file segment normally and continues execution.

4.2.5 Using a Parallel File Descriptor

There are four ways to use a parallel file descriptor. They consist of reading or writing fixed length records, reading **unknown** length records, writing **unknown** length records, and movement of the file pointer within the file. While Chapter 3.1.2 and Chapter 3.1.3 discuss **unknown** file segments in more detail, **unknown** length records mean that the actual size of the record read or written is known only after the individual I/O function is finished.

For each of these different types of I/O functions, there is a corresponding modification to the way the function behaves. In all cases, the first decision is to test if the file pointer supplied is considered a parallel or sequential file descriptor. If the file pointer does not have parallel behaviour, the sequential function is executed and the results are returned to the user.

In the case of reading or writing fixed length records, since the size is already known, the run-time system can verify that there is sufficient space available for the operation. Figure 4-9 shows the pseudo code for a parallel `fread`. The file pointer must first be determined to have a parallel behaviour (`Pilot_resolveAccess`). Then, sufficient space must be available to do the I/O operation (`Pilot_preVerifyFP`).

While exceeding the boundary is an error for **newspapers**, exceeding the boundary

```
int par_fread( char *ptr, int size, int nitems, FILE *stream )
{
  int status ;
  if ( Pilot_isParallel( stream ) ) {                /* Parallel I/O */
    Pilot_resolveAccess( stream ) ;
    Pilot_preVerifyFP( stream, size * nitems ) ;
    status = Pilot_fread( ptr, size, nitems, stream ) ;
    Pilot_postVerifyFP( stream ) ;
  } else {                                           /* Sequential I/O */
    status = fread( ptr, size, nitems, stream ) ;
  }
  return status ;
}
```

Figure 4-9 — Wrapper code for parallel `fread`.

for a **report** normally involves getting access permissions for the new segment. (The two exceptions for the **report** template occur when the new location is less than the value of the **base** of the first segment or the **extent** of the last segment is exceeded.) After performing the actual I/O operation (in the case of the **report**, this may consist of a read or write operation for each segment), the run-time system verifies that the file pointer is left in a consistent and valid state for the parallel template constraints, and that the file is updated in a consistent and reliable fashion (`Pilot_postVerify`). That is, temporary files or memory buffers are flushed to the master file on disk.

When the length of a read operation is not known until after the operation completes, two approaches are used to determine if the post-read state of the file is legal. Using `fscanf` (Figure 4-10) as an example, the I/O operation is allowed to proceed after access is permitted (`Pilot_resolveAccess`) but the post-I/O check (`Pilot_postVerifyFP`) will confirm that the I/O operation has been completed within the limits described by the I/O template. For global and independent I/O templates, this causes few problems. End-of-file (EOF) conditions will apply normally.

```
int par_fscanf( FILE *stream, char *fmtString, va_arg )
{
  /* The term va_arg indicates variable numbers of arguments. */
  int status ;
  if ( Pilot_isParallel( stream ) ) {              /* Parallel I/O */
    Pilot_resolveAccess( stream ) ;
    status = Pilot_vfscanf( stream, fmtString, va_arg ) ;
    Pilot_postVerifyFP( stream ) ;
  } else {                                         /* Sequential I/O */
    /* vfscanf is the va_arg equivalent of fscanf */
    status = vfscanf( stream, fmtString, va_arg ) ;
  }
  return status ;
}
```

Figure 4-10 — Wrapper code for parallel `fscanf`.

Unfortunately, post-read state check is an unsatisfactory solution for segmented templates since memory locations can potentially be modified when they should not be. An example of this would be where a read operation crosses the segment boundaries and gets data from the neighbouring segment. This could happen if PI/OT segments the file logically but not physically. The read should either read up to the segment boundary or fail completely because the exclusive access condition has been violated. By replicating a known length I/O segment locally, the EOF condition is exploited for the **newspaper** template and a solution is derived. **Unknown** segment lengths do not have this problem since they can use the conventional EOF to determine the limits on the file.

For **report** templates, the end-of-segment (EOS) condition is not the same as the EOF condition. An alternate solution is to parse the format string and test each read operation separately — a pre-read operation. If a pre-read operation crosses the segment boundary, the run-time system will need to exchange the segments with the file manager. However, the actual read operation will not fail or notice the exchange. The drawback to this system is the necessity of two read operations: one from disk and one from memory.

If a write operation does not provide a length until after the operation completes, there is a "simple" solution. Figure 4-11 shows the pseudo code for a parallel `fprint`. After access permissions have been resolved (`Pilot_resolveAccess`) for the parallel `fprintf`, the output is redirected to a temporary buffer (`tmpStream`). This could be a temporary scratch file or a memory buffer. Its length will be checked by the verify operation

```
int par_fprintf( FILE *stream, char *fmtString, va_arg )
{
  /* The term va_arg indicates variable numbers of arguments. */
  int status ;
  if ( Pilot_isParallel( stream ) ) {              /* Parallel I/O */
    FILE * tmpStream ;
    Pilot_resolveAccess( stream ) ;
    tmpStream = Pilot_tempStream( stream ) ;
    status = Pilot_vfprintf( tmpStream, fmtString, va_arg ) ;
    Pilot_postVerifyFP( stream ) ;
  } else {                                         /* Sequential I/O */
    /* vfprintf is the va_arg equivalent of fprintf */
    status = vfprintf( stream, fmtString, va_arg ) ;
  }
  return status ;
}
```

Figure 4-11 — Wrapper code for parallel `fprintf`.

(`Pilot_postVerifyFP`) and committed to the physical disk as necessary. If the parallel behaviour is global or independent, the memory stream returned is the actual stream. For a **newspaper** behaviour, if the buffer exceeds the segment size, the I/O operation is completed up to the boundary (if known) or normally (if unknown). Otherwise, the **report** behaviour causes a segment swap and the new file segment is updated with the remainder of the memory buffer.

Finally, there are the control I/O operations such as `fseek` (Figure 4-12). These functions change the location pointed to by the file pointer. In this case, after the access permissions have been resolved (`Pilot_resolveAccess`) the parallel seek (`Pilot_fseek`) is performed. The alternative `fseek` is necessary since the user sees and addresses a global or unified file. A segmented file must have the offset values modified to fit within the physical constraints of the file segment.

A post condition check (`Pilot_postVerifyFP`) will confirm that the file pointer is correctly updated. Seeking outside the fixed limits of a segment for a **newspaper** template is considered an error. **Unknown** file segments are only a problem if the file pointer is moved to a position less than the **base** value of the segment. This is considered an error condition similar to when a user attempts to sequentially access data before the beginning of a file. For **report** behaviours, exceeding the segment boundaries will cause the appropriate segments to be exchanged with the file manager except for the starting segment and the segment containing the EOF. In the case of the starting segment, it is an error to seek before the base value and seeking past the EOF only extends the segment if the length is unknown.

```
int par_fseek( FILE * stream, long offset, int mode )
{
  int status ;
  if ( Pilot_isParallel( stream ) ) {              /* Parallel I/O */
    Pilot_resolveAccess( stream ) ;
    status = Pilot_fseek( stream, offset, mode ) ;
    Pilot_postVerifyFP( stream ) ;
  } else {                                         /* Sequential I/O */
    status = fseek( stream, offset, mode ) ;
  }
  return status ;
}
```

Figure 4-12 — Wrapper code for parallel `fseek`.

4.3 PI/OT Template Implementation Issues

The previous section looked at the implementation issues from the viewpoint of the functions that are parallelized. This section examines the concerns of implementing the five template abstractions. There are three subsections that present specific implementation issues of the independent **photocopy** template, the three global templates (**log**, **meeting**, and **report**), and the two segmented I/O templates (**newspaper** and **report**).

4.3.1 Photocopy Template

The independent template, **photocopy**, treats files similarly to sequential stream I/O except that write operations are visible only to the local client. When the client is finished processing the file, the manager gets the updated file. The user-specified order of the write operations determines when changes to the manager's file become visible to the collective. If a process closes, opens, or reopens a file, the manager will only be informed when the client is finished processing. This can affect future usage of the file pointer for any **inactive** entries in the **call-chain** list, as they have not yet been sent to a remote process. However, this will not affect any of the **active** entries as they are executing concurrently.

4.3.2 Global Templates

The global file pointer templates (**meeting**, **log**, and **report**) have I/O stream behaviour similar to the sequential behaviour. There are differences when `fclose` and `freopen` are done by a client or when a group `fopen` occurs. Closing a file on the client side causes the manager to invalidate all remaining non-active I/O requests left on the **call-chain** list for that particular I/O object. Reopening the file results in all subsequent I/O access through the new file. As discussed earlier, when a collective `fopen` is done, one process is designated the manager to control access to the file pointer. The ordering attribute for the template (Chapter 3.2) defines which process gets access to the file next.

4.3.3 Segmented Templates

The segmented file pointer templates (**report** and **newspaper**) differ in that a client receives access permissions for a file pointer that lies within a range specified by the **base** (or starting point) in the file and the **extent** (or the number of bytes) that define the limit of the segment. At run-time, the base and extent for the client are determined by the manager, either through a user-supplied constant or a call-back segmentation function. The manager advances its file descriptor to point to the next byte after the end of the client's segment.

The segmentation function has a specific signature defined for it. Figure 4-13 shows an example function where a variable length record consists of an entry defining the number of elements followed by the elements. The first parameter is the parallel file stream (`stream`). A user assumes that the file pointer is set to the start of the record. The three remaining parameters are the minimum (`min`), maximum (`max`), and current (`current`) number of processes sharing access to this parallel I/O object. The PI/OT run-time system will invoke this function automatically. The PI/OT user-interface needs to ensure that this signature is used.

Note that the user writes this segmentation function using the standard stream I/O functions. The user is not permitted to write to the file in the segmentation function, but read and seek operations are permitted. The reason for this is that the segmentation function is only intended to examine the file, not modify it. The return value is the number of bytes composing the record. The usual parallel constraints still apply to the file descriptor used for segmenting. For example, access permission to the file must be granted; the file pointer must stay within the specified boundaries of its segment. (Recall that a segment can be segmented.)

```
#include <stdio.h>
#include <myTypeDefs.h>
unsigned long segmentFcn( FILE * stream, int min, int max, int current )
{
  unsigned long offset ;
  int nElements, status ;
  /* The number of elements composing this record */
  status = fread( &nElements, sizeof(int), 1, stream ) ;
  if ( status != 1 )                                /* The read has failed */
    return (unsigned long)-1 ;
  /* Calculate the number of bytes in this record */
  offset = sizeof(int) + nElements * sizeof(Element_typedef) ;
  return offset ;            /* Return the number of bytes in this record */
}
```

Figure 4-13 — An example of a PI/OT segmentation function.

In the example depicted in Figure 4-13, the function reads in the number of elements (`nElements`) found in the record. If there is an error in the read (no more data), the function returns the equivalent of negative one (-1) to the run-time system, indicating that an error has occurred. Otherwise, the size of the record is calculated and returned to the run-time system. The user does not have to restore or move the location of the file pointer prior to returning, as this is the responsibility of the PI/OT run-time system.

An alternate way of segmenting a file would be to have a record consisting of data on three lines (i.e. every third `new-line` character delimits a record). Figure 4-14 shows an example of such a segmentation function for PI/OT. The drawback to this approach to segmenting is that the data file is effectively read twice, once by the segmenting process and once by the client process.

The client uses a local copy of the segment if the file is opened in write or update mode. This local copy is not necessary if the file is opened using read-only mode. However, checking that the boundary conditions are not violated will require extra care if variable length read operations are used. If fixed-length records are specified, only the modified segments need to be returned to the manager for updating the master file. If the segment has not been modified, it does not need to be updated.

If a **newspaper** template that uses a **defined-length** extent (a value greater than zero) is selected, each process must stay between the two limits **base** and **base + extent**. **Defined-length** segments do not mean constant-length records. Rather, the extent is de-

```
#include <stdio.h>
unsigned long segmentFcn( FILE * stream, int min, int max, int current )
{
  /* A record is composed of three lines of data (delimiter character "\n") */
  unsigned long offset = 0;
  int count = 0, status ;
  while ( ! feof( stream ) && count < 3 ) {      /* Three line feeds or EOF */
    if ( (status = fgetc( stream ) ) == '\n' )
      count++ ;                              /* Another line feed encountered */
    offset++ ;                                          /* Another byte offset */
  }
  return offset ;                          /* Return the new file segment extent */
} ;
```

Figure 4-14 — Another example of a PI/OT segmentation function.

termined in advance of the remote process using the file pointer. Currently, the file is segmented either by a user-supplied constant or by a call-back segmentation function at run-time. An extension of this work would have the precompiler derive the segment size by analyzing the code. The user specifies **unknown-length** records by defining an **extent** of zero (0).

The **unknown-length** record size is useful when the file is opened using write-only mode. The processes write in distinct file-segments that are reintegrated into the file in consecutive segments. In read-only mode, the processes share the same base and, when all the processes have finished, the maximum of all the **extents** is used to update the parallel file pointer.

In update mode, the situation is more complicated. The approach taken is to have the processes read from the file and write to a local temporary file segment with the appropriate synchronization mechanisms. Reintegration depends on the ordering attribute. Concatenating separate segments, as is done with the write-only mode, is not appropriate. The intent of the update mode in either sequential or parallel applications is to modify existing data. Overlaying the segments in a user-specified manner (**ordered**, **relaxed**, or **chaotic**) respects this intent. However, if one process is reading from the file while another process is updating the file, the result of the read operation is indeterminate.

There are two solutions that avoid this non-determinism. The first is that any write operation must require the writing process to get an exclusive write lock on the portion of the file affected, prior to proceeding. The other processes can grant the write lock when they are able. This approach works adequately if the read-to-write ratio is large. However, getting permissions for every write operation is expensive.

The second solution defers the permissions phase until a process is finished with the file segment. The processes would read from the master file and write to a temporary file. All updates are made to the local copy. When the process is finished, it returns the updated segment to the manager. The manager then seeks the appropriate permissions from all the processes sharing this file pointer prior to updating the file with the modified segment. This is trivial when using **ordered** updates in that the process update order has been predefined. With **relaxed** or **chaotic** ordering, the update permission must be acquired from all participating processes. Depending on the application, the non-determinism inherent in this delayed update approach may not be acceptable.

One side effect of using **unknown-length** extents is that both testing for end-of-file or any read/write operation by anyone other than the owner(s) of the segment will block until the outstanding segments have been processed and reassembled in the file.

The order attribute indicates how the file will be reassembled when a client is finished with a segment. For example, if **ordered** I/O is specified, the segments are integrated as specified by the call ordering. If **relaxed** I/O is used, segments representing similar work (α type in Figure 3-2) are assembled in an as-received order with the other segments (β type in Figure 3-2) blocked from being committed to disk until all α segments are finished. **Chaotic** I/O allows any segment to be re-integrated into the manager's file in an as-received fashion.

When a **report** I/O operation crosses a segment boundary (either less than **base** or greater than **base + extent**), the client requests permission from the manager to access the new segment. The manager waits until the requested segment is free or it asks the process that currently has the requested segment to temporarily give the requested read or write permission for the segment to the client. If the segment is free, the manager passes the new segment on to the client. To prevent deadlock, the client gives up its current segment before receiving the new segment. The solutions proposed for **newspaper** templates with **unknown-length** extents are equally relevant for **report** templates.

To ensure that multiple processes do not have access to the same segment, the manager is responsible for preventing the client or calling process from attempting to move the file pointer into areas of the file still controlled by **active** segments in the **call-chain** list. Similarly, the manager will block the calling process on an `fclose` until all the outstanding segments are consumed. Client processes closing segmented files would not block but may have side-effects on the application. That is, closing a file will invalidate all non-active work left in the **call-chain** list for that file descriptor. If the work is invalidated, the next client process would be required to open a new file and have the manager re-segment it. There is no requirement to segment the new file at this point for every remaining element using this file in the call-chain; as the remaining elements are processed, they would be re-segmented.

The `freopen` does not affect processes currently working on an active segment but any outstanding segments waiting for a process are modified to use the new file. Again, the new file would be segmented upon request.

When a client finishes with a file segment, it sends a message to the manager that it is done. If the segment has been modified, the message also contains the modified segment. The manager processes the client's message and updates its file appropriately.

4.4 PI/OT and Enterprise

This section presents the specific details about the modifications made to the Enterprise PPS to support the PI/OT parallel programming model. Enterprise meets the minimum requirements needed to implement PI/OT as defined in Section 4.1. Enterprise has a well-defined graphical interface (GUI) for defining relationships between distinct computational tasks. A source-to-source translator (**precompiler**) is used to generate wrapper code for remote function invocation, synchronization points, and maintenance of futures. (A *future* [16] is defined as a memory location that is promised a value in the future. The process is allowed to continue processing until that memory location is accessed. If the value has not been received, the process blocks until the value is received.) The run-time system is responsible for spawning processes, ensuring messages are reliably sent between processes, marshalling and demarshalling data for remote function invocation.

The Enterprise GUI is used to maintain an external file that describes the parallelism for the application. In this file, the parallel tasks are identified and the relationships to the different tasks are defined. An Enterprise parallel task or **asset** corresponds to a function. Parallelism is realized when multiple invocations of a function are running concurrently, using several processes. The Enterprise PPS does not require the user to learn a new library for parallel behaviours, nor does it extend the sequential programming language (C). Enterprise is responsible for marshalling and demarshalling the parameters of asynchronous remote function calls and identifying *futures* or synchronization points in the user's code. These responsibilities are accomplished by a combination of compile-time analysis which identifies futures and remote function invocations; run-time libraries deal with the dynamic nature of the application. For more details about the Enterprise programming model as it pertains to this dissertation, see Appendix A. Detailed discussions about different components of Enterprise are found in [44, 45, 49-52, 64, 65, 70, 72, 73, 84].

There are three areas that required intervention or modification to the Enterprise PPS to support parallel I/O. They are the graph file, the source-to-source translator (**precompiler**), and the run-time library. Section 4.4.1 describes the changes to the graph file; Section 4.4.2 documents the modifications to the precompiler and associated scripts for the static analysis; Section 4.4.3 provides details about the modifications to the run-time libraries to support PI/OT.

There are a number of limitations to the current Enterprise implementation of PI/OT.

1. No direct advantage is taken of any physical parallelization of files (for example, striping or declustering). Rather, all files are treated as having a unified single logical image.

2. Enterprise uses the scope of a parallel function (**asset**) to define the lowest form of parallel activity. The definition of an I/O transaction is bound by this same scope. That is, arbitrarily sized atomic I/O statements are not supported.
3. The **report** template is not implemented yet. Neither is the merging of **photocopy** template writes. The deadlock prevention mechanism is not sophisticated enough to support these templates. More work is needed in the compiler portion of PI/OT to ensure deadlock does not happen.
4. For reasons similar to those given in 3, the collective open is not fully implemented. Deadlock prevention requires more compiler support to ensure a general solution.
5. There is no check if the user has multiple file pointers opening the same file. If the user opens the same file using file pointer `fa` with a global behaviour and a second file pointer `fb` with segmented parallel behaviour, the implementation does not link the two separate file pointers to the same physical file. Undefined results are expected.
6. The Enterprise precompiler examines only the source code files of the different *assets* that compose the parallel computations. The non-asset user source files or libraries are not yet searched by the precompiler to modify any `fopen` statements. Consequently, it is illegal to open a parallel file pointer in anything but a parallel function source file. The definition of parallel file descriptors and their behaviours is not yet resolved for non-asset user code. One solution is to query the process to find out which parallel task is being currently run. Extending the tuple information to include the sequential function as well as the variable name and current parallel task to identify the parallel I/O object may be one approach.
7. There is no parallelization of the `fdopen` function because the original low-level open function does not take into account possible parallel semantics. The two functions, `sscanf` and `sprintf` are not parallelized since they modify memory locations, not physical files. A block of shared memory would be considered the same as a file for these two functions. There is no reason why they cannot support parallel I/O semantics. When Enterprise supports distributed shared memory, a re-evaluation of the parallel behaviours possible for these two functions is needed.

4.4.1 Graph File Modifications

Each Enterprise asset type has its own parallel designation. The parallel I/O model needs to understand its own parallel requirements and the assets it will call to have the correct behaviour. In the Enterprise graph file, each asset definition contains a field that, for historical reasons, is unused. PI/OT uses this previously unused field, **EXTERNAL**, to define parallel I/O for each asset as a series of tuples. Modifications to the parsing of the graph file reflecting this change were made in the precompiler and run-time system. For now, the GUI was not modified as it is not used by PI/OT. Normally, the graph file data would not be written by the user. Rather, the GUI would create the text file based on all the information supplied by the user. For other PPSS, there must be some way of integrating the parallel task definitions with the parallel I/O requirements. Figure 4-15 shows the general outline of a PI/OT tuple's format for an Enterprise graph file.

Each tuple consists of the variable name of the file pointer used in the asset (`varName`). Following it is the parallel mode of the file pointer (`parMode`). There are currently five acceptable choices. The template read and write ordering attributes (`readOrder` and `writeOrder`) can be optionally defined for the parallel mode. For read ordering, they are `ro` (ordered read), `rr` (relaxed read), and `rc` (chaotic read). Similar attributes are specified for write ordering. The remaining mandatory entry defines whether the I/O transaction size (`BlockFactor`) is either a per-I/O statement (**a**`tomic`) which is currently ignored, or a per-

```
varName parMode readOrder writeOrder BlockFactor [assetName=segmentSize]

where:

varName       := character string
parMode       := {MEETING | LOG | PHOTOCOPY | NEWSPAPER | REPORT}
readOrder     := {ro | rr | rc}
writeOrder    := {wo | wr |wc}
BlockFactor   := { a | b }
assetName     := character string
segmentSize   := {0 | >0 | functionName}
functionName  := character string
```

Figure 4-15 — Format of a PI/OT entry for an Enterprise graph file.

asset (`b`lock). Since Enterprise does not permit missing entries, all of these mandatory entries must be present.

If the parallel mode chosen is segmented (`NEWSPAPER` or `REPORT`), there must be a segmentation function or constant (`segmentSize`) defined for every asset that gets passed the file pointer. The segmentation of a file is based on the requirements of the different computational blocks that access it. If there are different types of computational blocks, each type could require different amounts of data. If the asset is replicated and it invokes an `fopen`, `freopen`, or `fclose` function (collective behaviour), there must be a segmentation value specified for the asset. The value is either a zero for an **unknown** length segment, a positive integer representing the size of the **extent** in bytes, or a user-defined function that the run-time system will call to determine the **extent** whenever the file is opened or passed in a remote asset call. This function may also return the value of -1 (or its equivalent as an `unsigned long`), which indicates to the PI/OT run-time system that an error has occurred. The current action upon encountering a segmentation function error is to ask the PPS to report the error and shut down the parallel application.

To clarify the use of the PI/OT tuple in Enterprise, an example graph file is provided in Figure 4-16. The graph file is based on the example first discussed in Chapter 1.1. How this example has been "Enterprised" is explained in Appendix A.2.1. There are two parallel file pointers defined in the `Parent` asset stanza. The first one, `fin`, is to be treated as a segmented parallel file (**newspaper**) while the second one, `fout`, is a global write append parallel file (**log**). The granularity of atomic I/O operations is at the function block level rather than at the atomic statement level.

The read and write attributes are ordered for `fin`. This is the default case and there is no need to change this as `fin` is opened read-only. However, the write ordering of `fout` can be relaxed since it was determined that the order of output was irrelevant as long as transactions did not overlap each another. In this case, chaotic writes would be equally acceptable since only `Child` processes access the file.

```
Parent  line  1  1  1  ORDERED  NODEBUG  NOPTIMIZE
CFLAGS
EXTERNAL  fin  NEWSPAPER  ro  wo  b  Child=4  fout  LOG  rr  wr  b
INCLUDE sherwoodpk
EXCLUDE maligne-lk
Child  individual  3  4  UNORDERED  NODEBUG  NOPTIMIZE
CFLAGS
EXTERNAL
INCLUDE
EXCLUDE sherwoodpk
```

Figure 4-16 — An Enterprise graph file with PI/OT extensions.

The `Parent` process opens the file. Since it cannot be replicated (a property of being the first asset) there is no need for a definition of a segmentation value. However, `Parent` does pass the two file pointers to `Child`. In this case, the user chose to limit each segment size to a constant value of four bytes and the definition is `Child=4`. An alternative would be to define a function (`Segment4`) that returns the value 4. In that case, the definition would be: `Child=Segment4`. There is no need to specify the segmentation functions in a particular order, nor to specify any unused segmentation functions.

4.4.2 Static Analysis Additions

The Enterprise source-to-source translator (**precompiler**) uses the graph file to modify only certain parts of the user's source code for the parallel computational behaviour — the asset source files. However, PI/OT needs to replace all the stream I/O functions except for `fopen` in both the asset code and non-asset source (sequential) code. Enterprise does not examine sequential code because there is no control parallelism located there. Because this deficiency (from PI/OT's viewpoint) of the current Enterprise precompiler not examining all the source code, a `sed`[2] script is used to search the user's sequential code and replace the I/O statements. One limitation of the `sed` script is that non-asset code cannot open a file using a parallel behaviour since the signature of the `fopen` is not changed. Because the precompiler looks only at the asset source code, only the asset code has the `fopen` function calls modified.

PI/OT replaces the standard stream I/O functions with a corresponding parallel I/O stub function, `ENT_XXX`, where the `XXX` is replaced with the name of the corresponding I/O stream function or macro. This replacement is done before the source code is preprocessed either by the Enterprise precompiler or by the conventional compiler (`cc`). These compilers process any macros such as `feof` or `getc`. Each time the file pointer variable is passed in the I/O function invocation, the stub I/O function examines the pointer to determine if the variable is a parallel file pointer.

The lookup list (**ParIO**) for identifying parallel I/O descriptors is created and maintained by the PI/OT run-time system for each asset object controlled by a process. A query from the stub I/O function asks the run-time system for the asset currently in control. This asset is then asked to search its **ParIO** list to determine if the file pointer has a parallel behaviour.

The interesting part of modifying the source code is found in the entry points into the stream I/O. The two entry points are the `freopen` and `fopen` functions. As discussed in Section 4.2.3, there are two possible implementations of the parallel `freopen`. The first method, where the physical file is changed but the parallel I/O behaviour remains the same, was chosen. The `freopen` function signature is not modified and the `sed` script is sufficient to search and replace for the appropriate PI/OT stub function. This method allows the user to reopen a file in the sequential code as the parallel behaviour has already been established.

The other user entry point is `fopen`. Figure 4-17 shows the modified function signature. In this case, the file pointer has not been initialized nor has the file been identified as a parallel file. The run-time system needs to couple parallel behaviours to files. Recall that a parallel computation task is already identified by Enterprise as an asset. Consequently, any file opened by an asset may exhibit parallel behaviours. Adding the asset name allows the run-time system to know where to search for parallel information. However, a given asset can have several file pointers, each with different parallel behaviours. The name of the actual variable that is being assigned is passed to the run-time system to ensure that the correct parallel I/O object is updated.

---

[2] sed (stream editor) is a standard UNIX program that applies a series of editor commands to a file.

```
FILE * ENT_FOPEN( char *fileName, char *mode, char *variableName, char *assetName )
```

Figure 4-17 — The signature of an Enterprise parallel `fopen` function.

The Enterprise precompiler was modified to search the asset code files for occurrences of `fopen`. The precompiler then adds two text strings to the `fopen` parameter list. The first text string contains the name of the assigned variable in the `fopen` statement. The second text string contains the name of the asset which identifies the transaction type. All `fopen` functions in asset source files are modified, regardless of whether they are designated parallel or not. If a file pointer is not designated as having a parallel behaviour, the pointer is treated as having a sequential behaviour. Thus, a user need only modify, add, or delete file pointers in the graph file and rerun the application again to test a new parallel I/O behaviour.

It is important for the parallel `fopen` function to return a normal stream I/O variable. A user calls sequential functions using the file pointer. If a parallel file pointer is passed to sequential functions, the correct parallel behaviour is still performed, as all the I/O functions in the source code have been replaced with parallel equivalents. A possible improvement would be to replace the sequential I/O library with a parallel version. This parallel library would only be linked if the application has been properly preprocessed to reflect the modified parallel I/O function signatures.

The Enterprise precompiler also modifies the stub or wrapper functions it generates for each type of asset invocation and the corresponding return variable so that the I/O variables are identified and declared properly. When a remote function is invoked, the parameters are sent to the process that will execute the function. If the message contains file pointers, PI/OT replaces those file pointers with the correct parallel I/O data structures. Upon receipt of the message, the remote function would create the parallel file pointers before invoking the function. When the remote function returns, the modified file pointer is intercepted and the appropriate parallel I/O data is sent.

The precompiler also adds the names and addresses of the segmentation functions for each asset into a global vector so the run-time code can look up a given function by name and then invoke it. Some work has been done to search the symbol table of the executable so that this global vector is unnecessary. However, if the executable is stripped (i.e. the symbol information is removed), such a search does not work.

4.4.3 Run-time Libraries

The PI/OT run-time I/O library, while based on Enterprise functionality, is not dependent on Enterprise. The ability to send and receive messages and retrieve parallel computational information is all that is required. There is some intervention in the Enterprise run-time code to include the parallel I/O functionality which occurs in five places: the asset graph (parallel I/O data management), the remote function invocation (file pointer marshalling), processing (file pointer demarshalling), the remote function return (updating and integration of file), and parallel I/O event generation. The actual parallel I/O code was kept separate from the existing run-time library as much as possible.

The Enterprise implementation of PI/OT has modified the graph file to store the static parallel I/O information. At run-time, the graph file is converted into an **asset graph**. This graph contains both dynamic (the number of processes for a given parallel type, which process manages the group) and static information (type of parallel behaviour, constraints about process location) for each parallel computational type. For parallel I/O, each parallel object (or asset) maintains the two parallel I/O lists, **call-chain** and **pending**, plus a description and current status for each parallel I/O object assigned to the asset (the **ParIO** list).

Each entry in the **call-chain** and **pending** list contains sufficient information to describe and construct a parallel I/O object. Each entry contains the address of the manager of the I/O descriptor, the type of parallel I/O behaviour, the name of the file currently attached to this descriptor, the **base** and **extent** (if segmented), the access permission, ordering information, and the internal stream file pointers.

In some cases, the parallel behaviour requires two separate file descriptors – one for use by the application and one for any temporary files or memory buffers. Temporary files are located by default in the directory `/tmp` to minimize the impact on the network. This default value is overridden if the user sets the shell environment variable, `ENTERPRISE_IOTEMP`, to indicate some other location. For example, setting the environment variable to `tmpIO` will locate any temporary files in the directory `tmpIO` which is located relative to the current working directory of the application.

All stream I/O functions have parallel counterparts except for `sscanf`, `sprintf`, and `fdopen`. This makes the checking for parallel behaviours simple and consistent for the system. The actual check is a minor cost compared to doing the I/O. When the user invokes an I/O function, the parallel function determines if the I/O operation is for a parallel or sequential file pointer. If the file pointer is sequential, it is passed to the appropriate sequential I/O function and the return code is passed back to the user. If the file pointer is parallel, its asset is queried for the current state of the parallel object. Depending on the actual I/O operation, the currently defined parallel I/O behaviour, and the state of the asset, a number of messages are generated to prepare for the I/O operation. The run-time system checks the I/O operation to confirm it can be safely done within the constraints of the parallel template and then performs the I/O operation. Afterwards, the return code of the I/O function is passed back to the user.

Each invocation of a remote function call that includes a file descriptor in the formal arguments passes the current state of the parallel I/O object. This state is stored in the **call-chain**. The Enterprise marshalling code was modified to detect the file descriptor parameter. The Enterprise precompiler modified the stub codes. The run-time system searches for a file pointer in the asset's **ParIO** list. The parallel I/O object is updated and flattened into an ASCII stream suitable for sending to the remote process. If the parallel behaviour is segmented, the I/O segmentation function associated with this parallel I/O object and the called asset is invoked to determine the **extent** of the segment.

Each function invocation is assigned a unique (per process) identifier which doubles as the transaction identifier. The **call-chain** list controls the sequence or progression of I/O operations for the application. As the remote assets (**client**) reach the point where they need to update or gain access to a particular I/O object, the caller asset (**manager**) regulates which process gets access to the file or file segment.

When a client receives a parallel file object, it creates a new I/O object based on the data stream received. Since a client can have multiple assets, the I/O object must contain the name of the asset it is associated with. The client then matches the object's asset type to one of the assets it manages. The matched asset then searches in its **ParIO** list for the parallel I/O object. The asset's I/O object is updated and the file is opened appropriately. If the run-time system fails to find the correct asset and the correct I/O object, the application is shutdown. When the asset parameters are completely processed, the computational function assigned by the user to the asset is called. When the file pointer is accessed in the user's code, the access permissions are required from the caller asset prior to performing the I/O. A message is sent to the I/O manager to request access which blocks the client asset from further processing until the manager returns.

The caller asset may receive many such requests and saves the ones it cannot satisfy in the **pending** list. For example, if a request is received out of order (this depends on the ordering attribute) or the caller does not currently have access permission, the request is queued. The caller asset periodically checks through the **pending** list to see if it can resolve any outstanding I/O requests.

As each asset finishes with its file pointer, it updates the parallel file data structure, closes the file pointer, and returns the modified file pointer to the caller. Upon receipt, the caller searches its **call-chain** list and deletes the corresponding object after updating its own parallel file object in the **ParIO** list, if necessary. This is different from the Enterprise model of parameter passing which allows parameters to be considered as one of three types — `IN`, `OUT`, or `INOUT`. `IN` parameters are not returned to the calling asset and consequently do not generate futures. `OUT` parameters are only returned to the calling assets and are not passed to the remote asset. `INOUT` parameters are passed in and returned back. The last two parameter types always generate a future. All I/O objects are considered as being `INOUT` parameters and will always generate a return message containing data. However, I/O objects do not generate a future in the Enterprise sense. The checking of the current I/O state by the parallel I/O stub functions replaces the generic future checking done by Enterprise. The marshalling code reflects this approach regardless of any attempt by the user to override (e.g. using the Enterprise `IN` or `OUT` marshalling macros).

In addition to the parallel I/O objects contained in **ParIO**, each asset definition is associated with a parallel I/O behaviour — **single**, **managed**, or **replicated**. Depending on the behaviour, the I/O is performed differently. For example, if a parallel `fopen` call occurs in an asset that has a **replicated** I/O behaviour, the asset would request its I/O manager to coordinate the open. Alternatively, a **managed** asset is a specialized system object that contains no user code. Its purpose is to synchronize and coordinate access to a file. These behaviours are based on the asset's replication factor and whether the asset is defined to be a manager or worker type by the Enterprise runtime system.

As a tool for performance monitoring and debugging, Enterprise generates event messages for parallel activities. These messages are generated and collected when debugging the application or analysing for performance bottlenecks. Since I/O can have a significant effect on the performance of a parallel application, a set of I/O events was made available to the Enterprise interface. A list of the current I/O events is found in Figure 4-18. They are typically in pairs.

An I/O transaction starts when the I/O manager sends a message (`sentIOMsg`) and the client receives it (`rcvdIOMsg`). When the client actually starts the transaction, the event, `processIOMsg`, is generated. If the client is finished using a given parallel file pointer before the overall I/O transaction is finished, the `doneIOMsg` event is generated. This indicates that the client has no further use for this file pointer but does not necessarily indicate that the I/O transaction is completed. That is, the run-time system can determine (or be prompted by the user) that the indicated parallel file descriptor will not be used any more in

```
#sentIOMsg manager client msgTag time IOState
#rcvdIOMsg client manager msgTag time IOState

#processIOMsg client manager msgTag time IOState

#doneIOMsg client manager msgTag time IOState
#sentIOReply client manager msgTag time IOState
#rcvdIOReply manager client msgTag time IOState

#IOAccessSurrender client manager msgTag time IOState
#IOAccessGained manager client msgTag time IOState

#IOAccessRequest client manager msgTag time IOState
#IOAccessGranted manager client msgTag time IOState

#performIO process IOType msgTag time IOState
```

Figure 4-18 — The different I/O events for Enterprise.

the transaction. This early release can improve concurrency. Currently, the early release mechanism is inserted by hand. Future research with the static analysis should do this automatically. The manager processing the reply generates the `rcvdIOReply` event. The end of an I/O transaction generates two events — the client indicates it is done (`sentIOReply`) and the manager processes the reply (`rcvdIOReply`).

If a client surrenders the access permission of a given file pointer to its I/O manager, the `IOAccessSurrender` event is generated with the corresponding manager's `IOAccess-Gained` event. The `IOAccessRequest` event is sent by the client to the manager for permission to access the requested file descriptor. Both global and **report** templates can generate this request. When access is granted, the manager generates the `IOAccessGranted` event. The manager generates a `rcvdIOReply` event upon processing the early-release.

The format of the message for any of these paired events is the same. The message consists of a tag identifying the event, a parallel task identifier of the event generator (this is unique for the application), the parallel task identifier of the I/O operation recipient (again unique for the application), a per-process unique message tag, a time stamp vector (for ordering of events), and the current state information of the parallel file pointer.

For the one unpaired event, `performIO`, the format of the message consists of the event tag identifying the event, the unique parallel task identifier of the event generator, the name of I/O function performed (for example, `fseek`), a per-process unique message tag, a time stamp vector (for ordering of events), and the current state information of the parallel file pointer.

4.5 Deadlock Prevention

A PI/OT implementation must address the issue of deadlock [19, 39, 68, 71, 74]. With the sharing and coordination of multiple file pointers tied together by remote function invocation or by collective open statements, there is a significant component of any implementation that is concerned with deadlock prevention. The PI/OT run-time system must determine if a request for a given file pointer is safe or deadlock-free.

First, the four necessary conditions for deadlock are listed. The PI/OT constructs associated with each condition are identified. Second, measures to avoid or eliminate the conditions for creating deadlock are described. Finally, the current state of deadlock prevention in the Enterprise implementation of PI/OT is presented.

There are four necessary conditions for deadlock [19]. They are:

1. **Mutual Exclusion**. The global parallel I/O behaviours (**meeting**, **log**, and **report**) satisfy this condition since only one process can access a file or file-segment at a time. This condition cannot be completely eliminated.
2. **Hold and Wait**. This condition occurs if a process holds a lock and is waiting for exclusive access to another file that is locked by another process. If a transaction requests all its locks prior to execution, this condition is avoided. However, utilization will drop if all the locks are not needed immediately. To increase the concurrency in a PI/OT transaction, the locks are not all sought after at once. Rather, asking for exclusive access or a file lock is delayed until the first I/O access in the function. If the order of requesting file locks is not consistent, this condition will be satisfied. The **relaxed** and **chaotic** ordering attributes of PI/OT templates ensure that this condition cannot be avoided without committing to exclusive access for all the other global file pointers in the transaction.
3. **No Preemption.** A process with exclusive access cannot have access taken away until it has finished all the I/O with that file pointer. I/O access is asked for when needed. A premise of PI/OT is that I/O cannot be rolled back. This condition for deadlock cannot be eliminated.

4. **Circular Wait.** There exists a cycle of processes each waiting for a resource the next process holds and will not release. This case is similar to the situation in Condition 2. Transactions are identified by the remote file pointer variables. Cycles could inadvertently be created if some unique method of identification of a file pointer is not used. The current deadlock prevention implementation uses the unique per-machine, low-level operating system file number to identify parallel file pointers rather than the name or address of the file pointer structure in order to avoid this aliasing problem.

When is deadlock prevention not needed in PI/OT? If there are no global I/O behaviours in the transaction, the application avoids the first deadlock condition. Having only one global file pointer per transaction avoids the second and fourth deadlock conditions. If the read and write attributes for all global file pointers are defined as **ordered**, there is no deadlock. The order of access to the file has been pre-defined and any out-of-order requests will be held until needed. Relaxing the ordering attribute of any one of the global file pointers (**relaxed** or **chaotic**) can lead to deadlock.

4.5.1 PI/OT Deadlock Prevention In Enterprise

The Enterprise view of the scope of a transaction is defined as the scope of a remotely executed function. However, a transaction does not start until a remote process actually uses one of the parallel file pointers. When a process requests access to a global file pointer, the entire transaction that the remote process is using is examined and all global file pointers contained in the transaction are then committed to that process. This ensures that the hold and wait condition is not possible. All the necessary file pointers will eventually be accessed. This is a first step to eliminating the potential for deadlock. However, the ordering attributes can create a condition where deadlock is possible.

When a process is seeking exclusive access to a particular file pointer, the current deadlock prevention algorithm first determines the overall ordering of the transaction that includes the file pointer. This is based on all the global file pointers in the transaction. The run-time system identifies the critical file pointer which could cause a deadlock condition. If this is not done, deadlock can occur.

Consider two global file pointers, `f` and `g`. I/O operations with `f` are defined as **ordered** while with `g` the ordering is **relaxed** or **chaotic**. If a process, $\mathbf{P_t}$, generates three of these transactions, the pending list would contain the following entries (f_1, g_1, f_2, g_2, f_3, g_3) where the subscript indicates the transaction identifier and italicized entries indicate that access is available to be granted.

If $\mathbf{P_t}$ grants a requesting process, $\mathbf{P_2}$, access to g_1 (a property of the ordering semantics of `g`), the transaction tuple is marked as owned by $\mathbf{P_2}$. The pending list looks like (f_1^2, g_1^2, g_2, f_3, g_3) where the superscript entries indicate the identifier of the process that now owns the transaction. The bolded entries indicate that this process currently has access. However, this is incorrect as f_1 should not be granted to $\mathbf{P_2}$ since access to `f` is defined as **ordered**.

The current solution defines an overall transaction ordering attribute based on the most conservative value of the attributes of the parallel file objects. Using the first example, the request from $\mathbf{P_2}$ for `g` would not be granted as the ordered attribute indicates that only $\mathbf{P_1}$ has the correct tag in order to grant access to f_1. When any file pointer of a marked transaction is ready to grant access, the permission is sent to the marked process which may or may not be blocked waiting for it. Still, other processes waiting on the file pointer are blocked until the marked process relinquishes the access permission.

Part of the conditions for granting access is validating the file pointers. To show the necessity of this, consider three file pointers, f_1, f_2, and f_3 that are connected to three different files. The three file pointers are managed by one process and are used in two different transaction types. The first transaction definition contains f_1 and f_2 while the second transaction contains f_1 and f_3. With a **chaotic** ordering defined for `f`, a process of the second transaction type could ask for access to f_1 and inadvertently be granted a transaction of the first type. This results in an error.

The current solution to deadlock prevention limits the expressibility of the ordering attributes. More work is needed in this area. As seen later on in Chapter 5.3, an early release mechanism can be used to avoid the need for deadlock prevention and subsequently improve performance. Future work with static analysis can automatically insert early release functionality into the user's code.

4.6 Chapter Summary

This chapter outlined **how** the PI/OT programming model should be implemented in general and how PI/OT has been implemented in the Enterprise parallel programming system. It is important to note that only minimal intrusion into the PPS is necessary to implement this system. This minimal intrusion bodes well for transferring PI/OT to other parallel programming systems. The efficiency of the Enterprise implementation is dealt with in Chapter 5.

There are few changes to the standard stream interface. Only the `fopen` function has had its signature changed and the changes can be handled automatically by a source-to-source translator. The various I/O macros, such as `getc` or `feof`, have been replaced with functions to allow a test for parallel behaviours. The user specifies **what** computational and I/O parallel behaviours are required, separate from the source code. The compiler then uses this information to determine **how** to modify the source code in order to implement the parallel behaviours. The run-time system takes the dynamic information and determines **how** to perform the I/O efficiently.

For the Enterprise implementation of PI/OT, the precompiler replaces the stream I/O functions with matching wrapper functions in the asset source code and, in the case of `fopen`, adds two variables to the parameter list. For the non-parallel source code, a `sed` script was used which does not modify any function signatures. This last step means that only the asset source code can open a parallel file descriptor.

There are five identified areas where the run-time library of the PPS interacts with the PI/OT run-time library. They are: the asset graph (parallel I/O data management), the remote function invocation (file pointer marshalling), processing (file pointer demarshalling), remote function return (updating and integration of file), and parallel I/O event generation.

The only non-standard user-level feature of this implementation is the use of call-back functions for the dynamic segmentation of the file. Later implementations may try to have the compiler provide this information. For static segmentation, the user can specify a constant value in the external specifications. Changing the value does not force a recompilation of the code because Enterprise reads the parallel specifications at run-time. Since the compiler modifies all the I/O statements, the only time an application needs to be recompiled is when the segmentation function is changed. Otherwise, adding, modifying or deleting parallel templates is a run-time operation. This makes for rapid prototyping of the application.

If more than one file pointer is shared between the various processes, deadlock prevention is needed. The scope of a transaction is the scope of the remote function. Consequently, the first parallel file pointer used will set the ownership of the entire transaction. The ordering attributes of individual file pointers can cause deadlock. However, determining an overall ordering attribute for the transaction based on the most conservative ordering will avoid this potential for deadlock. The cost of deadlock prevention is seen in the lack of expressibility of the ordering attributes and the subsequent reduction in concurrency of the application. If it is not safe to grant access, the remote process is blocked.

Chapter 5

5. Performance

The description of the model (Chapter 3) and the implementation (Chapter 4) have been presented. This chapter describes the attempts to justify the claims that this top-down model to parallelizing I/O is simple and effective to use while providing a reasonable performance when compared with the current hand-coded approaches.

Five experiments are presented. The first two address performance comparisons between a hand-coded approach using PIOUS and the high-level PI/OT approach. The two applications display a similar computational parallelism but have quite different I/O requirements which stress the I/O system(s). The first has fine-grained I/O and the second has coarse-grained I/O.

The first performance experiment (Section 5.1) is drawn from a molecular docking application at the University of Alberta. The original application looks at placement and alignment of a protein fragment onto a larger protein molecule (the smaller fragment "docks" at the larger molecule). The application used in this experiment consists of reading, processing, and writing blocks of data (molecules) on disk. These blocks consist of objects within objects within objects. Each object can be of variable length on disk. Each individual I/O operation is quite small, consisting of four to several hundred bytes within a single record. The second performance experiment (Section 5.2) consists of coarse-grained I/O — disk-based matrix multiply. The application processes a large (1.3 gigabytes) amount of data and quickly saturates the network.

The third set of experiments, Section 5.3, examines the useability and composability of PI/OT as discussed in Chapter 3.2. This experiment consists of two parallel computational approaches (heterogeneous children, Sections 5.3.1 and 5.3.2, and a pipeline, Sections 5.3.3 and 5.3.4). Synchronization between different types of child processes is required and is based on the run-time invocation behaviour (the **call-chain**, Chapter 4.2). As well, the inheritance of the caller's I/O constraints modify the child's subsequent use of the file pointer in any remote procedure call. Both the performance and useability of some of the different combinations of the PI/OT I/O model and the Enterprise parallel programming model are presented.

By specifying the parallel computational and I/O requirements separate from the source code, no recompilation is required when the I/O templates are changed. The effects of increasing process replication factors or changing the I/O model are examined. The Enterprise programming model does require recompilation if parallel computational tasks are changed. That is, if two separate task types that were specified as parallel are now considered as one, or if one task type is split into two, the wrapper functions must be reimplemented by the Enterprise precompiler. Another part of this third set of experiments examines the potential gain in concurrency (if any) by the timely release of file access permissions. Currently, the insertion of the release mechanism is done by hand. However, future work will look at the insertion of the release mechanism by compiler tools.

The fourth experiment, Section 5.4, revisits the molecular docking problem first presented in Section 5.1. This time, the effect of dynamic segmentation, where the size of each file segment is determined at run-time by using a call-back function, is examined. The performances of three different segmentation approaches are examined using four levels of computational granularity.

In the fifth experiment, Section 5.5, the lessons learned in the previous experiments are applied to the fine-grained example first seen in Section 5.1 The effect on performance is examined when a more complex parallel computational and I/O version is created. The ease of integrating the additional parallel I/O requirements with the new computational parallelism is shown. Performance results indicate that this more complex application can yield better performance than the simpler versions discussed in Section 5.1 and Section 5.4 when using a heterogeneous workstation cluster instead of the alternative homogeneous workstation network. Section 5.6 presents a summary of this chapter.

The Enterprise parallel programming system [70] was used to implement PI/OT. The PIOUS parallel I/O system [57] was used for performance and coding comparisons. The choice of PIOUS was made for three reasons. First, the MPI-IO implementations which had just been released at the time of testing were alpha implementations based on a changing "standard". It would be difficult to draw meaningful comparisons and conclusions about the performance of either system. Second, PIOUS has been available for over a year and seems relatively stable. The third and primary reason for using both Enterprise and PIOUS for these experiments is that these two systems both use PVM [30] as the underlying communication system.

Previous work [65] examined the performance of Enterprise and PVM as it relates to the cost of communication and templated computational parallelism. The simple parent-child computational parallelism of both performance experiments provides similar performance with either Enterprise or PVM. This provides a comparison point to ensure that the current implementations of Enterprise and the hand-coded PIOUS applications are performing adequately. By keeping the hardware and the communications software constant, more meaningful comparisons can be drawn about the I/O.

Comparing PIOUS with template I/O is not intended as a critique of PIOUS or of any other parallel I/O system. Rather, it is intended as an experiment to see if parallel I/O templates are viable. It is assumed that low-level libraries and special parallel file systems like Galley [61] would be integrated with the high-level templates in a fashion similar to what Enterprise has demonstrated with computational parallelism.

For all of these experiments, the parallel times given are the elapsed times or clock-on-the-wall times. The sequential times, unless otherwise noted, are the user times as determined by the `getrusage` system function and represent the actual time spent by the application using the processor.

5.1 Fine-grained I/O

This section examines in detail the parallelization of a real problem in order to illustrate that template I/O can realize little or no loss of performance in comparison to the implementation in PIOUS. The program is derived from a molecular docking problem in biochemistry at the University of Alberta. The original application looks at placement and alignment of a protein fragment onto a larger protein molecule (the smaller fragment "docks" at the larger molecule). Each molecule and fragment is stored as a series of nested objects on disk. That is, one object contains other objects which in turn contain other objects. Each object is has a variable size.

For this experiment, the biochemistry component was removed and replaced with a function that simulated the computational time spent on each sub-object. This allowed more control and flexibility when changing the computational granularity. The reading and writing of objects is dispersed throughout the computations and is fine-grained in nature (four to several hundred bytes).

In Figure 5-1 the application specifics have been abstracted out, leaving the high-level I/O view of the program. The code looks similar to the example given in Chapter 1.1, but the `Child` function is different. As well, the `rewind` introduces new synchronization considerations.

```
#include <stdio.h>
main( int argc, char **argv )
{
  FILE *fin, *fout ;                      /* Input and output file descriptors */
  fin = fopen( argv[1], "r" ) ;                        /* Open the input file */
  fout = fopen( argv[2], "w+" ) ;                     /* Open the output file */
  while ( ! feof( fin ) ) {                         /* Until end of file, work */
    Child( fin, fout ) ;
  }
  fclose( fin ) ;                                     /* Close the input file */
  rewind( fout ) ;              /* Rewind the output file to the beginning */
  Stats( fout ) ;                     /* Perform summary statistics on output */
  fclose( fout ) ;                                   /* Close the output file */
  return 0 ;
}
```

Figure 5-1 — Sequential code for fine-grained I/O test program.

In the sequential version, the `Child` reads data from a file (`fin`) and performs calculations, with the results going to an output file (`fout`). Once the input is exhausted, the main program rereads the output file to analyze the results (`Stats`).

5.1.1 Data File Layout

The input and output files contain data objects within data objects within data objects. Each object has its own specific read and write functions and knows how many immediate sub-objects it contains. All I/O is spread throughout the code and is quite fine-grained (four to several hundred bytes at most for any individual I/O operation). In the real application, the data objects are all variable length. In order to make it easier to compare performance with PIOUS, the objects were fixed in size with the resultant input records set to a constant length of 352,108 bytes, creating output records 18,050 bytes in length.

The format of the input file is such that an arbitrary number of `Child` records are stored consecutively. Only by reaching the end of the file does the application know how many `Child` records are in the file.

A `Child` input record (Figure 5-2) starts with a four byte integer, `n`, indicating the number of `CEDE` record blocks. A `CEDE` block (shaded) consists of one `C` block, one `E` data block, one `D` data block and one `E` data block. The `n` `CEDE` record blocks follow. After these data blocks, a single `E` record block (shaded) indicates the end of the `Child` record.

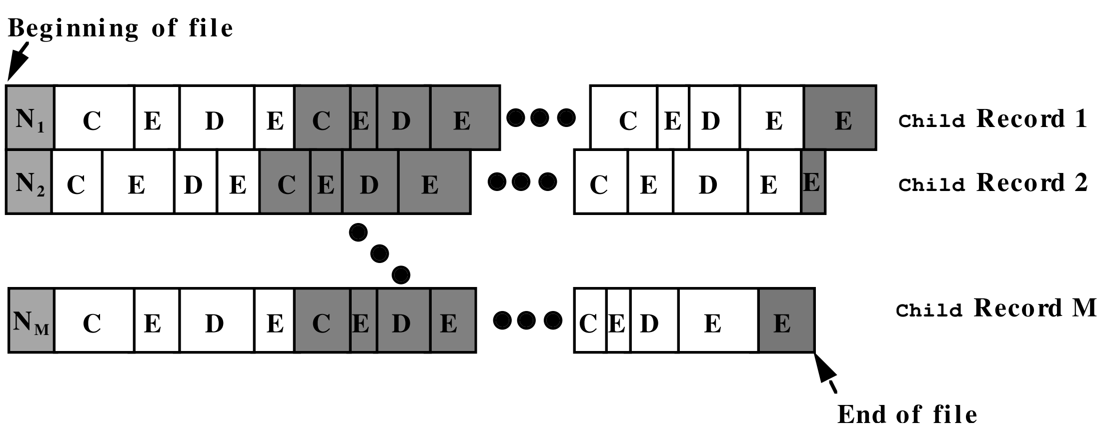

Figure 5-2 — Layout of an input data file for the fine-grained I/O experiment.

A `C` and `D` record block have similar formats on disk. They consist of a four byte integer which indicates the size of the two vectors that compose the balance of the record block. There is first an integer vector followed by a character vector. An `E` record block is similar to the `C` and `D` record blocks except that the order of the two vectors is reversed (i.e. a character vector followed by an integer vector).

The output file consists of a number of `Child` output records. The actual number depends on the number of input records processed. The output file of a `Child` record starts with a single character indicating the type of record block, followed by a four byte integer indicating the number of `CEDE` output data blocks that follow. Each `CEDE` data block is composed of a `C`, `E`, `D`, and `E` record block. These records have the same format, consisting of a single character indicating the type of record, a four byte integer for the number of elements used, and a constant sized vector of `doubles` (eight byte real numbers). A single `E` record block indicates the end of the `Child` output record. The size of a `Child` output record is dependent on the number of `CEDE` records read in from the input file.

5.1.2 Parallel Design Considerations

Since the `Child` tasks are independent of each other, multiple `Child` processes can run concurrently. They need only coordinate reading from the input file and writing to the output file. There is no need to preserve the correlation between the input file order and the output file order.

Coordination of the input file must guarantee that each input datum is processed precisely once. Since it does not matter which `Child` does which piece of work, segmenting the input file avoids the inefficiency of having to synchronize file access. Each `Child` process reads a contiguous interval in the file. The program has been set to use an input segment size of 352,108 bytes. Output file access also needs to be synchronized. The sequential program appends to the end of the output file. However, since the output data is a fixed size for each piece of input data, the output file can also be segmented.

Segmenting both the input and output files eliminates the need for `Child` processes to synchronize their concurrent activities. However, they must synchronize before the sequential `Stats` function can be called. A barrier is necessary to guarantee that all the results are in the output file. The barrier is found in the `rewind` function since this function puts the `Parent`'s file pointer in a position that potentially allows two processes access to the same segment. `Stats` does a sequential read of the output file, summarizing each record. If the parallel application is created by hand, a parallel programmer must be careful with the output file, since the `Child` function will need to treat it as parallel I/O, while `Stats` will treat it as sequential I/O.

Since there are few constraints on the ordering of input and output, it allows experimentation with a variety of parallel I/O implementations.

5.1.3 Template I/O in Enterprise

Using the graphical interface, the programmer specifies that one process, called `Parent`, can call multiple instances of the `Child` process. To have this program run correctly under Enterprise, the user must make a number of small changes (modifying the `Child` parameter list and renaming the `main` function), as shown in Figure 5-3. The changes to the user code are Enterprise-specific (either for data marshalling purposes or identifying parallel tasks) and have nothing to do with parallel I/O. In the implementation generated by Enterprise, each call to `Child` is translated into a message sent to a remote process. The Enterprise run-time system takes care of the spawning of processes, communication (sending, receiving, marshaling/demarshalling of data), synchronization, and program termination.

```
#include <stdio.h>
Parent( int argc, char **argv )         /* Identify this as a parallel task */
{
  FILE *fin, *fout ;                    /* Input and output file descriptors */
  fin = fopen( argv[1], "r" ) ;                       /* Open the input file */
  fout = fopen( argv[2], "w+" ) ;                    /* Open the output file */
  while ( ! feof( fin ) ) {                       /* Until end of file, work */
    Child( fin, 1, fout, 1 ) ;              /* Data marshalling of pointers */
  }
  fclose( fin ) ;                                    /* Close the input file */
  rewind( fout ) ;                  /* Rewind the output file to the beginning */
  Stats( fout ) ;                      /* Perform summary statistics on output */
  fclose( fout ) ;                                  /* Close the output file */
  return 0;
}
```

Figure 5-3 — Modifications to sequential code for Enterprise.

The application parallelism is specified graphically in Enterprise and is saved in a file separate from the sequential source code (the *graph* file). Enterprise uses a source-to-source translation tool (**precompiler**) to insert the correct code to do message communication and synchronization. The translator has been modified to look for parallel I/O file descriptors (as identified in the graph file) and replace them with calls to parallel I/O functions. The machine-generated source code is then conventionally compiled and linked for a particular target architecture. The Enterprise run-time library uses the graph file and run-time computational behaviors to implement the parallel I/O operations. Since the I/O behavior is interpreted at run-time, the user can change the I/O templates without having to recompile the program.

For this example, Figure 5-4 shows the additions necessary to the Enterprise graph file to specify the **newspaper** template for the `Parent` process. This change reflects the fixed size input and output file segmentation used for comparison between the PIOUS and PI/OT implementations. The PI/OT modifications to the Enterprise precompiler ensure that all occurrences of these file pointers in `Parent` and `Child` will have the appropriate parallel I/O semantics enforced.

```
fin NEWSPAPER ro wo b Child=352108 fout rc wc b NEWSPAPER Child=18050.
```

Figure 5-4 — Modifications necessary to the Enterprise graph file for fine-grained I/O.

A **newspaper** (segmented file) requires a segment size. Figure 5-5 shows an example of a segmentation function appropriate for the input file pointer for this application. Ideally, this consideration should be transparent to the user but, unfortunately, it is difficult to automatically choose a good segment size since the user knows best how the I/O is to be accessed. For segmented files, PI/OT allows the user to provide a call-back function that specifies the segment offsets. In this example, the fine-grained nature of the file is clearly illustrated. Each object contains a header that provides sufficient information for the user to calculate the offset into the file for the next object. The total offset for the `Child` segment is returned to the run-time system.

Figure 5-6 shows an example of an I/O segmentation function for dynamic output records that relies on the **newspaper** semantics to append the **unknown** sized segments in the output file. Using a **report** instead would result in the merger of the **unknown** file segments, which is unacceptable since all the output data is needed, not just one record.

```
#define IC ( sizeof(int) + sizeof(char) )
#define I sizeof(int)
unsigned long InputSegSize ( FILE * fp, int min, int max, int current )
{
  unsigned long offset ;                            /* Size of this segment */
  int CD, C, D, E, i ;
  i = fread( &CD, I, 1, fp ) ;           /* How many CEDE records are there */
  if ( i != 1 )                         /* End of file or file error return error */
    return (unsigned long)-1 ;
  offset = I ;                           /* The record includes the size of CD */
  for ( i = 0 ; i < CD ; i++ ) {           /* Loop reading the CEDE records */
    fread( &C, I, 1, fp ) ;                      /* Elements in this C record */
    fseek( fp, C * IC, SEEK_CUR ) ;               /* Skip over the C record */
    offset += I + C * IC ;                       /* The size of this C record */
    fread( &E, I, 1, fp ) ;                      /* Elements in this E record */
    fseek( fp, E * IC, SEEK_CUR ) ;               /* Skip over the E record */
    offset += I + E * IC ;                       /* The size of this E record */
    fread( &D, I, 1, fp ) ;                      /* Elements in this D record */
    fseek( fp, D * IC, SEEK_CUR ) ;               /* Skip over the D record */
    offset += I + D * IC ;                       /* The size of this D record */
    fread( &E, I, 1, fp ) ;                      /* Elements in this E record */
    fseek( fp, E * IC, SEEK_CUR ) ;               /* Skip over the E record */
    offset += I + E * IC ;                       /* The size of this E record */
  }                                /* End of loop reading in the CEDE records */
  fread( &E, I, 1, fp ) ;                        /* Elements in this E record */
  offset += I + E * IC ;                     /* The size of trailing E record */
  return offset ;                        /* Return the extent for this segment */
}
#undef IC
#undef I
```

Figure 5-5 — An example I/O segmentation function for fine-grained I/O test program.

One of the advantages of the PI/OT approach is the ease with which a different I/O parallelism can be selected for the application. For example, the parallel template for `fin` can be changed from **newspaper** to **meeting** and the program immediately re-run without recompilation. As well, `fin` could be converted back to sequential I/O without any additional effort by the user. This makes it easy for the user to experiment with different types of I/O (and computational) parallelism. Note that in any other system, changing the I/O behavior would usually necessitate many changes to the source code.

```
unsigned long OutputSegSize( FILE * fp, int min, int max, int current )
{
  return (unsigned long)0 ;
}
```

Figure 5-6 — An example of an I/O segmentation function for dynamic output records.

5.1.4 PIOUS Implementation

Three classes of PIOUS implementations were built. All PIOUS applications must import a file into the PIOUS file system before the file can be accessed using the PIOUS library routines. Similarly, the output file must be exported back to the regular file system. A user needs to write these conversion routines.

The first PIOUS implementation class used global file pointers. Because ordering of the input and output file is not required for this application, the input and output files could be treated as globally shared resources. Globally shared files effectively have one global file descriptor, for which all processes have to synchronize their access. (This is similar to the **meeting** or **log** templates.) The program retrieved an entire data segment as one single block I/O operation and cached the block on local disk storage (default is `/tmp`). The locally cached data was processed using the conventional stream I/O with the output again going to local disk storage. The `Child` source code was not modified. After each `Child` function finished processing, the results were added to the PIOUS output file as another single block I/O operation. When the end of the input file was reached, each `Child` process notified the `Parent`. When all the children had reported in, the `Parent` continued on to the sequential part of the computation.

This approach proved to be the easiest to implement since most of the explicit parallelism was hidden by the global shared file synchronization. It allowed minimal impact on the existing user's code by using the standard I/O operations to read the local file and then create the output data segment.

A second implementation class involved importing the input file into PIOUS as a list of segments and creating a corresponding list of empty output file segments. (This is similar to the **newspaper** or **report** templates.) The user had to write additional code to distribute the input segments as they were requested by idle `Child` processes. Initially, the `Parent` process allocated one segment to each `Child`, but as a `Child` completed its work, the `Parent` was responsible for allocating it a new segment.

Each `Child` process opened the appropriate input and output file segments, copied the local segment of work to a temporary file in one I/O operation, opened the temporary output file, performed the work, and then exported the local output file back to the parallel output file (again in one operation). This repeated until all segments were distributed. The `Parent` process was then informed and the `Child` process exited after cleaning up the temporary files. The advantage of this method is that the output is in the same order as the sequential version. Again, the `Child` code was not touched.

The final implementation class was to write a pure PIOUS application using the PIOUS segmented file capabilities. However, instead of importing or exporting a block of work to local storage, all parallel I/O operations were identified and replaced with the appropriate PIOUS function calls. This was the most intrusive solution as significant portions of the `Child` code needed modifications.

Each of the three classes required a significant amount of new code. This would also be true when using any other low-level parallel I/O library. The first implementation class which cached data blocks locally using the global file pointers is given in Appendix B.1. (Note that much of the implementation has been abstracted into subroutines that, for brevity, are not included.) The original sequential version is about 530 lines of code; the parallel version is approximately 350 lines longer. Any changes in the I/O functionality of the program must be reflected in the source code. For example, if the user wants to do the equivalent of changing from a **newspaper** to a **meeting**, a considerable number of changes have to be made to the source code, with the resulting overhead of testing and debugging the changes.

Within the first two implementation classes, three versions were developed. The first version used the standard stream I/O functions without any changes. The second version modified the standard I/O stream to use large buffers (using the `setbuf` function). The third version replaced the standard I/O stream functions with low-level I/O functions (`read` and `write` instead of `fread` or `fwrite`). The third implementation class had only one version — all PIOUS calls.

5.1.5 Fine-grained I/O Performance

The testing configuration consisted of one Sun Sparc 4 (SS4), two Sun Classics, four Sun ELCs, and four Sun SLCs processors connected by a 10Mbps Ethernet. All processors had a local disk used for temporary files and swap. In addition, one Classic and the SS4 provided NFS file systems to the other processors. Unless otherwise stated, there was only one compute process per processor and all processes were assigned to the fastest available processor. The processing power of these different processors relative to the slowest processor type (SLC) is as follows: the SS4 is 3.0 times faster, the Classic is 2.4 times faster, and the ELC is about 1.7 times faster when running the same application. This is an approximation based on the performance of the applications presented in this chapter.

Three sequential versions were created. The first used standard I/O functions. The second version used low-level I/O functions to see if there is any performance improvement. The third version increased the `stdio` system buffer space (using `setbuf`) to see if there is any performance gain. The sequential user times when running on the fastest processor (SS4) using the local disk are given as: 1914 (standard), 1932 (low-level), and 1916 (buffered) seconds. The poorer performance of the low-level I/O can be attributed to the fine granularity of the read and write operations. Similarly, the performance of the buffered I/O is not significantly different from the standard I/O because of the fine granularity of the I/O operations.

A total of seven PIOUS versions were developed as described in Section 5.1.4. They are presented along with the segmented I/O (**newspaper**) Enterprise version. In most cases, little or no added benefit was seen for the extra programming effort.

The first PIOUS class uses global file semantics with local file caching of file segments (Global Stream PIOUS, or GSP for short). That is, a large PIOUS I/O operation is done and the resulting block is cached on a local disk. The user's code reads from this local file while writing to another local file. After the work is finished for this segment, the local output file is read in and written to the PIOUS file in one operation. The second class uses a similar approach to the first class except that the files are segmented by PIOUS rather than by the user (Stream Segmented PIOUS, SSP). However, the user is responsible for distributing access permissions to the remote processes for each segment. In the third class, all the I/O is done in a segmented file system using PIOUS function calls, without any caching (Pure Segmented PIOUS, PSP). From the programming perspective, this version required the most number of code changes.

Table 5-1 contains the results for the fine-grained I/O tests. For both systems, the time for starting PVM and for spawning the remote processes is ignored. The cost of the PIOUS import operation (30-60 seconds depending on the segmentation factor) is ignored as this could be considered a one-time cost if the input file was generated *in situ*. Similarly, the costs of creating and exporting the output file back to the network file system are ignored (5-10 seconds).

Child Processes	PI/OT	PIOUS						
		Global Stream (GSP)			Stream Segmented (SSP)			Pure Segmented (PSP)
		Buffer	Stream	Low	Buffer	Stream	Low	
2	1484	1315	1322	1276	1294	1409	1284	1917
5	813	703	699	705	737	704	705	1040
10	513	509	505	510	519	509	511	799

Table 5-1 — Elapsed times in seconds for PI/OT and PIOUS (PSP, SSP and GSP). PIOUS import and export times are not included. Sequential user times in seconds were: 1916 (buffered), 1914 (standard stream), and 1932 (low-level).

The parallel times presented are the best elapsed times of at least five runs. The processors and network used for this set of experiments were unavailable for exclusive use. The runs were collected over several weeks during times of quiescence but significant variance was seen. Automatic system functions such as backup operations appeared to cause significant interference.

Enterprise has one version that gives acceptable parallel I/O performance: both the input and output files are segmented using the **newspaper** template. Another version uses the **newspaper** template for the input and the **log** template for the output. This did not give good performance because the output file was locked until all the write operations for a given `Child` process were finished. As the write operations pervade the entire `Child` computation block, the other `Child` processes were quickly blocked waiting for access. The times for this inferior version are not shown.

The results show the effect of using two separate file systems for the physical storage of the global data files. (Recall that temporary files are stored on the local disk attached to each processor.) PIOUS is able to use two (or more) physical file systems to improve performance and concurrency. In the case of Enterprise, the input file was on one file system and the output file was on the other file system. PIOUS always distributed files between the two file systems. Wherever possible, the effect of the network was minimized. The number of processors used was one more than the number of children to account for the `Parent` process. No processor ran more than one process (`Child` or `Parent`).

The GSP version using global file pointers shows little difference from the SSP version. The PSP implementation uses PIOUS to perform a significant number of fine-grained I/O operations. This is very expensive as each I/O operation is converted to a message. This approach does show a performance gain over the sequential version but the gain is not as much as in the other implementations.

The PI/OT performance, although faster than the sequential version, was ten to sixteen percent inferior to the GSP and SSP versions. Even though it used a similar design in its implementation, the cost of using templates to abstract the parallel I/O diminished only with a larger replication of the workers. A likely reason for equality between the two systems is that the capacity of the network imposes an overall limiting factor. Ten processes asking for separate data blocks create a theoretical demand of three megabytes on a ten megabit network. As well, the file servers must get the data block on and off the physical disk for each requesting process.

Another factor for the poorer Enterprise implementation is that it checks every I/O operation if the file pointer has parallel behaviour. If there are many I/O operations, this cost becomes more significant. Clearly, for this example, there is a performance cost to using templates. Still, the Enterprise application shows improved performance compared with the sequential time. Future work on optimization using prefetching and compiler code analysis to order I/O operations should improve the template performance.

The benefits of templates are seen in the amount of modification to the user's code and the ease of changing parallel behaviours. Each PIOUS version took several hours to modify and debug. For the Enterprise version, the changes to the sequential code, as specified in Section 5.1.3, were done and the application was generated. This took about twenty minutes from starting with the sequential code until the first test run. The application was first tested using a **meeting** template for `fin` and a **log** template for `fout`. Performance runs were generated in **newspaper** mode simply by changing the parallel behaviour type for both file descriptors and making no changes to the code! No recompilation was necessary as the segmentation function was a constant size. Any performance penalty for using templates should be weighed against the potential benefits of quickly getting the parallel application up and running.

It is interesting that by using the global synchronization offered by PIOUS with the caching of input and output segments to allow stream I/O operations, this application shows the best performance. However, would this be the case if the application only does coarse-grained I/O?

5.2 Coarse-grained I/O

Disk-based matrix multiplication was chosen as the coarse-grained I/O application. This application is simple to code and can be done using coarse-grained I/O operations. The **A** and **C** matrices were segmented into user-specified stripes with the **B** matrix independently read by each processor. The **B** matrix was transposed on disk to improve data processing.

The sequential program (the source code for `main` is found in Figure 5-7) takes as command line arguments (`argv`) the names of the three files, the number of elements per row of the matrices and the number of rows per computational block. For simplicity, all three matrices are assumed to have the same rank. The `main` function opens the three files and until the end-of-file marker is encountered in the **A** file, it calls `Child` in a loop. The file pointer for the B matrix file is rewound after every call to `Child`. After the loop exits, `main` closes all three files.

```
#include <stdio.h>
main( int argc, char **argv )
{
  int NumberElements ;                      /* Number of elements per row */
  int BlockSize ;                             /* Number of rows per block */
  FILE *fA, *fb *fC ;                  /* Input and output file descriptors */
  fA = fopen( argv[1], "r" ) ;                   /* Open the A matrix file */
  fB = fopen( argv[2], "r" ) ;                   /* Open the B matrix file */
  fC = fopen( argv[3], "w+" ) ;                  /* Open the C matrix file */
  /* These two variables are used to partition the matrix into stripes */
  NumberElements = atoi( argv[4] ) ;  /* Convert from string to integer */
  BlockSize = atoi( argv[5] ) ;
  while ( ! feof( fA ) ) {                       /* Until end of file, work */
     Child( fA, fB, fC, NumberElements, BlockSize ) ;
     rewind( fB ) ;            /* Rewind to the start of the B matrix file */
   }
   fclose( fA ) ;                                /* Close the A matrix file */
   fclose( fB ) ;                                /* Close the B matrix file */
   fclose( fC ) ;                                /* Close the C matrix file */
   return 0 ;
}
```

Figure 5-7 — Sequential source code for matrix multiply `main` (Parent.c).

The function, `Child` (the source code is found in Figure 5-8), reads in the user-specified block of the **A** matrix as one read operation. The **B** matrix must be read in its entirety and is done so in a loop using stripes similar in size to those used to read the **A** matrix. The same sized stripes were used for programming simplicity. Each **A** and **B** stripe is multiplied together and stored in the appropriate location in the **C** matrix stripe. After the **B** matrix file is exhausted, the completed **C** matrix stripe is written to disk in one operation.

A benefit of using large I/O blocks sequentially is seen if non-blocking I/O is used to overlap I/O and computations; however, the code complexity increases. In this case, there is little benefit to using asynchronous I/O as the I/O buffer is needed immediately after the I/O call (the **B** matrix) and the size of the **B** matrix precludes it from being cached in mem-

```
void Child( FILE *fa, FILE *fb, FILE *fc, int nelems, int nblocks )
{
  double *A, *B, *C ;                              /* The A, B, and C matrices */
  int k, n, j, status ;
  /* Allocate memory for each block of A, B, and C */
  A = (double *)malloc( nblocks * sizeof( double ) * nelems ) ;
  B = (double *)malloc( nblocks * sizeof( double ) * nelems ) ;
  C = (double *)malloc( nblocks * sizeof( double ) * nelems ) ;
  /* Read in the block of A for this call to Child */
  status = fread( A, sizeof( double ), nelems * nblocks, fa ) ;
  if ( status < nelems * nblocks ) {                        /* End of file */
    return ;
  } else {                                   /* Do some work with the data */
    k = 0 ;
    while ( 1 ) {                                       /* Loop forever */
      /* Read in, one block at a time, all of B until the read fails */
      status = fread( B, sizeof( double ), nelems * nblocks, fb ) ;
      if ( status < nelems * nblocks ) break ;          /* All done here */
      for  ( n = 0; n < nblocks ; n++ ) /* Do the striped matrix multiply */
        for ( j = 0 ; j < nblocks; j++ )
          C[ n*nelems+k+j ] = DotProduct( &A[i*nelems], &B[j*nelems], nelems ) ;
      k += nblocks ;
    }                                               /* End of while loop */
    /* Write out the completed block of C */
    fwrite( C, sizeof( double ), nelems * nblocks, fc ) ;
  }                                     /* End if not end-of-file encountered */
  free( A ) ;                                   /* Free allocated memory */
  free( B ) ;
  free( C ) ;
  return ;
}
```

Figure 5-8 — Sequential source code for matrix multiply `Child` (Child.c).

ory at the processor. If double buffering is used, the overall in-memory capacity of the application is reduced by one quarter assuming that only the **B** matrix stripe is doubled up.

5.2.1 Parallel Design Considerations

The same computational parallelism used by the fine-grained I/O application in Chapter 1.1 and Section 5.1 was used. One of the differences is that this application has three parallel file pointers (the **A** , **B**, and **C** matrix data files) each with different behaviours. The **A** file pointer is treated as a segmented input file with each child process getting one segment or stripe of the matrix to read. The **C** file pointer is also segmented so each child process can write the corresponding answer stripe. The user or system must coordinate and preserve the relationship between **A** and **C** segments as an out-of-order **C** matrix is incorrect. The **B** file pointer is independent but is distributed to the child processes.

The other difference in this coarse-grained application is that I/O operations are few and can be quite large. In fact, with large numbers of processors and/or large matrices, the network will become the bottleneck depending on the physical disk layout in relation to the processors. This application will stress the network and file systems.

Observing the stress generated by this application is important if networks of general-purpose workstations are used instead of specialized hardware platforms or dedicated network farms. If parallel I/O is to be made easy-to-use by the "general" programming population, resource allocation and sharing become important — especially to system administrators.

5.2.2 Enterprise Implementation

Figure 5-9 shows the small number of modifications to the source code needed by the Enterprise version. The `main` function is renamed to `Parent`, extra parameters are added to invoke `Child`, and the `rewind` statement is removed. The `rewind` statement is not necessary in the parallel case as this function does not move the file pointer. The two file pointers `fA` and `fC` are segmented and need to have a segmentation function written for them. Note, in this case, one segmentation function can be used for both file pointers. The other file pointer, `fB`, is needed by all the `Child` processes but is considered independent as it does not require any synchronization.

```
#include <stdio.h>
static int NumberElements ;                    /* Number of elements per row */
static int BlockSize ;                           /* Number of rows per block */
Parent( int argc, char **argv )                  /* Identify the parallel task */
{
  FILE *fA, *fB *fC ;                    /* Input and output file descriptors */
  fA = fopen( argv[1], "r" ) ;                         /* Open the A matrix file */
  fB = fopen( argv[2], "r" ) ;                         /* Open the B matrix file */
  fC = fopen( argv[3], "w+" ) ;                        /* Open the C matrix file */
  /* These two variables are used to partition the matrix into stripes */
  NumberElements = atoi( argv[4] ) ;     /* Convert from string to integer */
  BlockSize = atoi( argv[5] ) ;
  while ( ! feof( fA ) ) {                             /* Until end of file, work */
    Child( fA, 1, fB, 1, fC, 1, NumberElements, BlockingFactor ) ;
    /* rewind( fB ) ; */                   /*This is not necessary in parallel */
  }
  fclose( fA ) ;                                      /* Close the A matrix file */
  fclose( fB ) ;                                      /* Close the B matrix file */
  fclose( fC ) ;                                      /* Close the C matrix file */
  return 0 ;
}
unsigned long AllMyIO( FILE *fp, int min, int max, int current)
{
  /* In all cases, return the same segment size */
  return (unsigned long)( NumberOfElements * BlockSize * sizeof(double) ) ;
}
```

Figure 5-9 — Enterprise code modifications to parallelize disk matrix multiplication.

The important modification to notice is the movement of the declaration of the two variables `NumberElements` and `BlockingFactor` from within the scope of `Parent` to being global only within the scope of the file (the `static` declaration). This permits the segmentation function to be declared in the file containing the `Parent` source code to permit dynamic segmentation of the data stripes. Of course, these two variables could have been declared global without the static limitation (a "free" global). Then, a separate file containing the segmentation function could have been used. However, creating a free global is not always possible or desirable in legacy code. As both stripes are equivalent in size and are the only file pointers to be segmented, the segmentation function simply returns the number of bytes composing one stripe.

To identify the three parallel file pointers in `Parent` to the Enterprise implementation, a number of changes are made to the graph file. A single line is added. While three lines are used for clarity in Figure 5-10, the graph file entry consists of only one line. The entry indicates that two of the file pointers (`fA` and `fC`) are to use the **newspaper** parallel behaviour (segmented) and the file segment size for both file pointers is determined by using the

```
fA NEWSPAPER rc wo b Child=AllMyIO
fB PHOTOCOPY rc wc b
fC NEWSPAPER ro wo b Child=AllMyIO
```

Figure 5-10 — Modifications to the Enterprise graph file for coarse grained I/O example.

segmentation function, `AllMyIO`. The file pointer, `fB`, is considered to be independent (**photocopy**).

5.2.3 PIOUS Implementation

The PIOUS version (the code is found in Appendix B.2) took about 375 extra lines of code to implement both the computational and I/O parallelism. The framework of the code is similar to the code needed for the fine-grained parallelism because the computational parallel behaviour is the same. The changes occur in the way the parallel I/O is handled. This implementation was quite intrusive and required modifications to the `Child` source code.

The reading of an **A** stripe and the writing of a **C** stripe are single I/O operations in the `Child` function. Converting them directly to PIOUS read and write functions saves the wrapper code from reading a stripe, caching it to local disk and then having `Child` re-read the local file. This is different from the fine-grained example where caching was beneficial. As well, the independent **B** matrix file is too large to cache locally. So, as PIOUS code would have to be inserted to read the **B** matrix anyway, replacing the UNIX `read` function for an **A** stripe and exchanging the UNIX `write` function of a **C** stripe operation for the equivalent PIOUS code is a minor addition.

An additional responsibility for the `Parent` is that now it must handle the coordination between the input and output file segments. The `Parent` process now manages the order in which segments are dispatched to the idle `Child` processes rather than letting the file system determine the order. Each `Child` works on one input and one output stripe. When no more stripes are available for processing, the `Child` process receives the no-more-work message (`segment=-1`) and then gracefully exits.

This application did not lend itself to any variations in the I/O parallelization strategies, as shown in the fine-grained example. Caching was not available and using global file pointers required synchronization between the input and output files. To do so results in a loss of computational concurrency.

In summary, this application was intrusive in respect to code modification for the parallel I/O in the user's source code. The user is responsible for coordinating access between corresponding segments as well as the import and export of file. However, the code framework for computational parallelism remained, for the most part, the same as the fine-grained example.

5.2.4 Coarse-grained I/O Performance

The testing configuration was the same as that of the fine-grained example in Section 5.1. The configuration consisted of one Sparc 4 (SS4), two Sun Classics, four Sun ELCs, and four Sun SLCs processors connected by a 10Mbps Ethernet. All processors had a local disk used for temporary files and swap. One Classic and the SS4 provided NFS file systems to the other processors. The processing power of these different processors is relative to the slowest processor type (SLC); the SS4 is 3.0 times faster, the Classic is 2.4 times faster, and the ELC is about 1.7 times faster when running the same application. This is an estimate based on several applications.

The parallel times presented are the best elapsed times of at least five runs. The processors and network used for this set of experiments were unavailable for exclusive use. The runs were collected over several weeks during times of quiescence but significant variance was seen. The automatic system functions, such as backup operations, caused significant interference.

Three sequential versions were created. The first one used the standard stream I/O. The second version used a buffered stream I/O. This buffering is done by using the `setbuf` function to direct the `stdio` system to create a single buffer that is large enough for the one data stripe. The third version used the low-level I/O functions instead of the stream I/O.

All three sequential versions had similar performance and were all within approximately six percent of each other. These experiments were run using the fastest processor and a local disk. The size of the I/O blocks was the same as in the parallel version. Not surprisingly, the buffered I/O (2214 user seconds and 125 system seconds) outperformed the other two versions. The low-level I/O version (2308 user seconds and 127 system seconds) was slightly better than the standard stream I/O (2352 user seconds and 142 system seconds). The high system time values for the standard stream I/O are the cost of the default buffer being two small which caused the system to thrash when transferring large blocks. To get the best performance, the program should either use no buffering or have sufficient buffer space to balance the cost of using it.

Table 5-2 shows the results for Enterprise and a purely PIOUS implementation multiplying two matrices of `doubles` (reals) stored in binary format and using a striping factor of 50 rows. Again, startup and the cost of importing and exporting the files into and out of PIOUS (180 seconds) is not included in the test results. Preliminary experiments with the 2000 by 2000 matrix showed that using a striping factor of 50 rows gave better performance than using 100 or 25 rows per stripe. The better performance is due to the ratio of work to message size and the different CPU speeds for the given network configuration.

The Enterprise results are better than those of PIOUS when using fewer child processes. This was unexpected but one explanation is offered. PIOUS uses direct process-to-process TCP/IP message-passing for parallel I/O, by-passing the PVM daemons. On the other hand, Enterprise uses both the network file system (on-demand messages) and default routing through the PVM daemons to communicate messages and file information. The performance differences can be attributed to the cost of using the TCP/IP instead of the UDP network protocols to transport data across the network. These differences are magnified by the amount of data being accessed (1,344 Mbytes). When the `Child` process is replicated ten times, PIOUS and PI/OT give comparable results. This is likely due to the network becoming saturated (measurements showed the network to be between 81% and 87% of maximum utilization).

These results point out that neither of these two parallel I/O systems can be considered as the best overall solution. Just because Enterprise uses the network file system, which in turn uses a different protocol for transmitting data, Enterprise performs better for this particular example. In contrast, the previous example shows that PIOUS performs somewhat better than Enterprise. The observed performance has little to do with the actual implementation of the I/O templates in Enterprise, but depends rather on the implementation of

Child Processes	50 rows per stripe	
	PI/OT	PIOUS
2	2225	2684
5	1473	1662
10	1598	1580

Table 5-2 — Disk-based matrix multiply elapsed times in seconds for 2000 by 2000 matrix of `doubles` (reals) using PI/OT and PIOUS (input and export times not included). Sequential user times are 2214 seconds for buffered stream I/O, 2352 seconds for stream I/O and 2308 seconds for low-level I/O.

the network file system. Nevertheless, templates once again yield comparable performance.

Ultimately, it is the network availability and capacity that determine the effect of I/O on the overall performance of a parallel application. By using different access patterns for the I/O, the requirements made upon the network and the file servers change the performance of the application. The ability to experiment with different parallel behaviours gives more flexibility in tuning an application to a specific network, processor, and data set. Templates offer this flexibility at little cost.

5.3 Useability and Composability

The two previous sections presented the performance advantages of the PI/OT programming model for a pair of applications that use the same simple computational model but have different I/O characteristics. The possible choices for parallel I/O were limited by performance considerations. This section examines the claims of the flexibility and software engineering advantages of PI/OT (Chapters 3) when more complex computational applications are developed. A pair of applications is presented. The first application parallelizes the I/O shared by a process and its heterogeneous children. (The heterogeneity is not found in processor differences but rather in the process differences.) The second application parallelises the I/O in a pipeline computation.

A heterogeneous child application occurs when one process type calls two or more processes types (Figure 5-11a). For example, a process of type **A** calls both **B** and **C** process types. There are an arbitrary number of the **A**, **B**, and **C** processes.

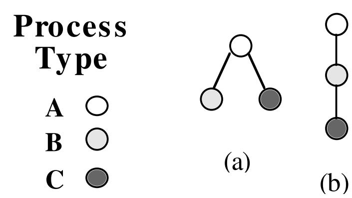

Figure 5-11 — Heterogeneous children and extended pipeline parallel computation configurations.

A pipeline application is used to demonstrate the composability of the I/O templates. In this case, **A** calls **B** which, in turn, calls **C** (Figure 5-11b). As each stage of the pipeline is encountered, different I/O abstractions can be imposed on the shared file pointer. Subject to constraints from earlier I/O decisions, more complex I/O descriptions are constructed and tested. As the computational parallelism is modified, the parallel I/O component adapts to the changes without intervention by the user.

For both of the applications presented in this section, the advantages of early release of I/O streams are examined. While this decision where to insert the early release is currently done by hand, future work with the source-to-source code translator would insert this release mechanism after analyzing the user's code.

5.3.1 Heterogeneous Children

If file pointers are shared between all processes, an I/O transaction is created and stored in the **call-chain** of the parent process as each remote process call is made. Parallel I/O may or may not require synchronization but coordination between processes is needed to access the file properly. If a global file pointer is shared, whichever process is active must have acquired a current and up-to-date version of the file pointer. With a segmented I/O file pointer, each process has clearly defined limits guaranteeing it exclusive access. Since the order of access to a file is not determined until run-time, the parallel I/O behaviour interacts with the parallel computational behaviour to coordinate access.

Depending on the I/O ordering attributes, synchronization of file access may be needed. In this case, the parallel I/O system is given a file access ordering and must block processes where necessary to preserve the ordering. Synchronization of file access forces a given process to wait until its predetermined (from the **call-chain** ordering) turn with the file. In general, this may reduce the level of concurrency available to the user's computations.

With the heterogeneous child parallel computational behaviour, an **A** process distributes work to the **B** and **C** processes. Figure 5-12 shows the source code for one such application. `Parent` corresponds to **A**, `Brother` corresponds to **B**, and `Sister` corresponds to **C** in Figure 5-11. The code is in the format suitable for an Enterprise asset code. The `Parent` process will open the input and output files. A step size, `n`, is supplied as a command line parameter by the user to segment the input file. The `Parent` process calls the remotely executed `Brother` function `n` times followed by a similar number of calls to the remotely executed `Sister` function. This pattern repeats until the input file is exhausted. The parameters to `Brother` and `Sister` are similar: a number representing the record tag being worked on, followed by two file pointers for the input and output files.

The Enterprise description of the computational parallelism is that `Parent` is a department asset containing two individual assets, `Brother` and `Sister`. Appendix A contains more details about the Enterprise nomenclature and programming model.

The program flow in the `Parent` source code reflects the structure of the input file which consists of `n` pieces of work for `Brother` processes followed by `n` pieces of work for the `Sister` processes. Using a **B** or an **S** to represent blocks of data on disk for a `Brother` or a `Sister` process respectively, input patterns such as **BSBSBSBS**, **BBSSBBSS**, **BBBBSSSS**, and so on, could be created. Depending on the input pattern and the computational load for each piece of work, different ordering attributes will provide different output patterns and different levels of concurrency. Depending on the user's requirements, the output ordering may be more important than maximizing the concurrency.

```
include <stdio.h>
Parent( int argc, char ** argv )
{
  FILE *fin, *fout ;
  int i, j, step ;
  fin = fopen( argv[1], "r" ) ;
  fout = fopen( argv[2], "w+" ) ;
  n = atoi( argv[3] ) ;
  i = 0 ;
  while ( ! feof( fin ) ) {
    for ( j = 0 ; j < n ; j++ ) {
      Brother( i+j, fin, 1, fout, 1 ) ;
    }
    for ( j = 0 ; j < n ; j++ ) {
      Sister( i+j, fin, 1, fout, 1 ) ;
    }
    i += n ;
  }
  fclose ( fin ) ;
  fclose ( fout ) ;
  return ;
}
```

```
int Brother ( int N, FILE *bin, int nin,
                          FILE *bout, int nout )
{
  int input, i, j;
  if ( feof( bin ) )  return 1 ;
  i = fscanf( bin, "%d", &input ) ;
  if ( i != 1 ) return 1 ;
  /* Early release of bin inserted here */
  Compute ( input ) ;
  PrintBRecord( bout, N, input ) ;
  return 0 ;
}
int Sister ( int N, FILE *sin, int nin,
                          FILE *sout, int nout )
{
  int input, i, j;
  if ( feof( sin ) ) return 1 ;
  i = fscanf( sin, "%d", &input ) ;
  if ( i != 1 ) return 1 ;
  /* Early release of sin inserted here */
  Compute( input ) ;
  PrintSRecord( sout, N, input ) ;
  return 0 ;
}
```

Figure 5-12 — Source code for the heterogeneous children example.

Equally relevant to improving the level of concurrency is the number of processes available to execute `Brother` or `Sister` as remote functions. Increasing the replication factor for either remote function should improve the computational concurrency. The number of consecutive **B** and **S** blocks make up the distribution pattern in the disk file. Figure 5-13 shows four possible connection patterns that could satisfy the parallel computational requirements.

The selection of a parallel I/O behaviour and ordering attribute is important since some choices can affect the computational concurrency by creating barriers in the code. Note the implied barrier in the `Parent` code (Figure 5-12) where the input file is tested if it is exhausted (`feof(fin)`). Imposing global file semantics on the input file pointer creates such a barrier whereas using segmented file semantics does not. Specifying ordered read attributes for the input file pointer imposes additional constraints on the potential concurrency since the application must proceed in the order of invocation rather than in the order of availability.

In Figure 5-13a, there is only one of each process type. The I/O requirements could be achieved by global semantics for the input and output. Unfortunately, using this approach is going to limit concurrency between the two child processes as each one will be waiting for the other to finish before returning both file pointers. However, inserting a function to release the input file pointer early allows the other process to read in the data and start computing. The code for both the functions `Brother` and `Sister` in Figure 5-12 has been commented to indicate the location in the code where an early release function for the input file pointers (`bin` and `sin`) could be safely inserted. If the input data is in the form **BSBSBS**, this will improve the overall concurrency. With only one of each process type available, other data patterns (for example, **BBSS** or **BBBSSS**) will not see much of an improvement except at the boundary conditions of the different data types.

The output file pointers do not benefit by an early release, as both functions return to the `Parent` immediately after calling the I/O function. Improved concurrency for the output file could be achieved if the order of output can be relaxed. Relaxing the ordering of the output file will see an improvement if the file pointer is locked until later on in the computations. If the computations have irregular granularities, the first process finished should be able to acquire the file lock. However, the current deadlock prevention mechanism commits both the input and output file pointers at the same time.

Segmenting the input or output file does not at first seem to be an appropriate step. However, the elimination of the barrier when checking for end-of-file in `Parent` allows several clusters of work to be generated. Using global semantics imposed a barrier with the `Parent` waiting until a single cluster of work is done before generating the next cluster. As well, prefetching of input data can improve the performance. Local caching of the data segment lessens the network demand for I/O. As seen earlier in Section 5.1, a large network I/O request followed by many small locally cached I/O requests is more efficient than many small network I/O requests. Early release of the input file will not affect concurrency much since both processes will be given distinct file segments and will consequently proceed independent of each other. However, early release could spread the network requirements for the update of written segments instead of having all updates occur at the end of the transaction.

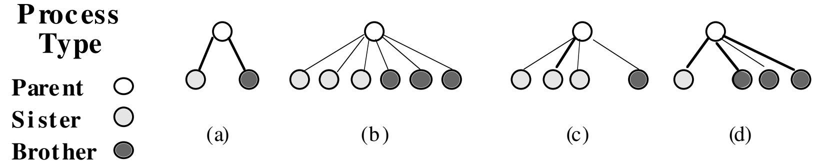

Figure 5-13 — Four computation configurations used for heterogeneous children example.

For the example application, segmented semantics on the output file pointer lead to the problem that the size needed for each output segment is indeterminable prior to the child processes starting their computations. One approach is to have the file segment set to some arbitrary maximum. This maximum size is dependent on the amount of temporary disk space available. Using some default value potentially leaves the output file with a series of holes. Alternatively, by using the **unknown** file segmentation, the file fragment could be merged back according to some predefined ordering attribute.

In Figure 5-13b, both the child processes (`Brother` and `Sister`) are replicated. Using global file pointers does not permit the extra processes to act in a concurrent fashion. In this case, early release of the input file will improve concurrency for data files with the format of **BBSSBBSS** or **BBBSSS**. Relaxing the ordering of the output file can now be done, either by clustering similar types of output data blocks in an as-received order instead of in an as-generated order, or by allowing chaotic ordering of the output by any process ready to write to the file. Using segmented I/O, the input file is divided up and each file fragment is prefetched as each computational process is assigned a given block of work.

Figure 5-13c and Figure 5-13d are mirror images of each other. All of the work can be done by one process type in a given time period, while replication of the other process type is needed to finish its work in the same time period. In these cases, using segmented file semantics for input is appropriate since all the replicated processes can start working. Eventually all the single process work is finished and the application will wait for the replicated processes to finish. Using global file semantics for the input with early release is appropriate for the replicated processes but it is not always appropriate for a single process type, except at the data borders. If the replicated processes access the file first, leaving the single process sufficient time to consume all of its work, this approach may work.

As seen, there are a number of choices the user has to make each time an application is run. If the data files are sufficiently varied, different versions of the application will be needed to efficiently process all the data sets. Specifying the parallelism separately allows the user to adapt the application to the data rather than having multiple applications.

5.3.2 Heterogeneous Children Performance

An example application using this computational model based on the code in Figure 5-12 was constructed. The two child process types, `Brother` and `Sister`, were run using three levels of computational granularity - fine, medium, and coarse. Both process types had the same computational granularity. The machines used in this experiment are all Sun4 ELC's with 12 megabytes of memory and a local disk for swap and temporary files. These machines are connected by a 10Mbps Ethernet network. All parallel runs reported are the average of five runs. The input data file consisted of 16 pieces of work laid out in the format **BBBBSSSSBBBBSSSS**. Sequentially, the application took 29, 79, and 232 user seconds to process the input file for each granularity level respectively. Thus, the average granularity for each block of work is approximately two (fine), five (medium), or fifteen (coarse) seconds of CPU time. This application is not I/O bound but the I/O does require synchronization.

Table 5-3 shows the results of a pair of experiments. The `Brother` and `Sister` processes were each replicated either two or four times, resulting in a total of five or nine processes for the computations (see Figure 5-13b). There were sufficient processors to ensure that there was only one computational process per processor. The shaded headings in the table indicate the type of template used for the input and output files. The application was compiled only once. All the parallel combinations were done by modifying the Enterprise graph file.

Within the range of values shown between different ordering attributes, there is little difference. The lack of difference can be attributed to the current deadlock prevention mechanism (Chapter 4.5) that limits concurrency. This mechanism uses the most conser-

File Pointers		Fine granularity		Medium granularity		Coarse granularity	
Input	**Output**	**Normal**	**Early**	**Normal**	**Early**	**Normal**	**Early**
Sequential		29		79		232	
`Brother` and `Sister` each replicated two times							
Meeting	**Meeting**						
Chaotic	Chaotic	50	27	102	49	265	130
Relaxed	Relaxed	51	32	104	58	266	141
Ordered	Ordered	51	32	104	58	266	140
Meeting	**Newspaper**						
Chaotic	Chaotic	43	19	95	43	257	118
Relaxed	Relaxed	43	18	94	42	257	118
Ordered	Ordered	44	18	95	43	258	118
Newspaper	**Newspaper**						
Chaotic	Chaotic	18		42		118	
Relaxed	Relaxed	18		43		118	
Ordered	Ordered	18		43		119	
`Brother` and `Sister` each replicated four times							
Meeting	**Meeting**						
Chaotic	Chaotic	50	24	102	33	263	79
Relaxed	Relaxed	50	26	103	37	264	84
Ordered	Ordered	50	26	102	38	263	84
Meeting	**Newspaper**						
Chaotic	Chaotic	43	11	95	22	256	60
Relaxed	Relaxed	43	9	95	22	256	59
Ordered	Ordered	44	10	95	22	256	60
Newspaper	**Newspaper**						
Chaotic	Chaotic	9		22		59	
Relaxed	Relaxed	10		22		60	
Ordered	Ordered	10		22		60	

Table 5-3 — Elapsed time (seconds) for three different parallel I/O template combinations, three granularity levels of computation, and two replication factors for heterogeneous children example.

vative ordering attribute of all the file pointers involved in the transaction. Consequently, when the input global file pointer is granted access permission, the output global file pointer is either granted permission or is promised access permission in an orderly fashion that avoids the potential deadlock. Both file pointers are released at the end of the entire transaction, not when the last I/O operation is completed. As a result, concurrency is unnecessarily limited.

A smarter compiler should be able to detect this independence and insert an early release for the file pointer. Now that the early release divides the transaction into two independent subtransactions, deadlock prevention is not needed. As well, asking for control of the first file pointer would not assign the second file pointer at the same time. This would lead to greater variety in the performance results and in the contents of the output file.

Part of this experiment examines the effect of an early release of a global file pointer. Early release is defined as safely releasing the file pointer when it is not needed any more in the calculations. The normal situation has all the file pointers released when the remote function completes its calculations. That is, release is done when the function returns. Using early release shows an overall improvement since concurrent computations are possible instead of waiting for the entire transaction to finish. However, the output file pointer is still committed at the same time as the input file pointer. The larger increase in performance using the chaotic ordering with early release is attributed to the removal of the restriction that all the `Brother` processes must finish before the `Sister` processes can proceed. Clearly, early release is beneficial to this application. However, the deadlock prevention mechanism hides any performance differences with the ordering attributes.

As the order of asking for the global file pointers does not change in this application and all global file pointers are guaranteed to be asked for, deadlock cannot happen. Future research in compiler support is necessary to analyse the user's code to determine if deadlock prevention is required at all. This static analysis could then advise the run-time system whether or not to invoke the deadlock prevention algorithm. The consequence of the current conservative and general approach to deadlock prevention is the observed reduction in concurrency and the uniformity of performance.

For this application, early release does not directly affect a segmented input file pointer as permission is already granted to access the file fragment and the file is not modified by the remote processes. However, if the file fragment had been modified, the reintegration of the remote segments into the master file could be affected depending on the defined write ordering attribute. Early release has the potential to even out the demand on the network by spreading out I/O messages instead of clustering them at the end of the computations of the remote function.

Segmented I/O for both input and output streams gives the best performance. The mixed case of global input and segmented output shows comparable performance to the segmented I/O only when the early release is used. Considering the ratio of the time spent reading in the data and the time spent computing, the early release result is not unexpected.

5.3.3 Extended Pipeline Example

The composability of the PI/OT templates is explored through a pipeline computation. The number of combinations available using a multi-stage pipeline grows rapidly. Consider that for each parallel file pointer there are the choices of five templates, three ordering attributes for read operations, and three ordering attributes for write operations. Currently, that gives the user forty-five (45) different combinations per file pointer. At each stage of the pipeline, except for the last one, all the parallel file pointers can be redefined. Obviously, not all these choices will make sense but the number of appropriate choices is still quite large. This exponential growth provides a rich set of choices for the user.

To clarify this growth, consider a two-stage pipeline that shares two parallel file pointers. The number of possible combinations for different I/O is $45 \times 45 = 2{,}025$. Adding an extra stage to the pipeline and sharing the file pointers in all stages increases the possible combinations to over four million. Alternatively, adding another parallel file pointer increases the two-stage pipeline parallel I/O combinations to 91,125 and the three-stage pipeline to over eight billion choices.

This section analyses the effects of parallelising the I/O for a three-stage pipeline mode. All three stages share a common input and output file pointer. Coordination and synchronization are needed at each level of the pipeline. Clearly, not all the I/O combinations will be presented here. Three pairs of parallel I/O behaviours are examined. A pair is defined as the parallel I/O behaviour assigned to the input file pointer and the parallel behaviour assigned to the output file pointer. Five different replication factors for the pipeline are studied using these three pairs. As well, the effect of early release of the input pointer in the last two computation stages is examined.

The first stage of the pipeline, `StageI` (Figure 5-14), opens the input and output files. Then, until the input file is exhausted, it calls the second stage of the pipeline, `StageII`, passing the current state of the two file pointers. After the input file is exhausted, `StageI`

```
StageI ( int argc, char ** argv )
{
  FILE *fin, *fout ;                /* The input and output file pointers */
  int i ;                                                  /* A counter */
  fin = fopen( argv[1], "r" ) ;                   /* Open the input file */
  fout = fopen( argv[2], "w+" ) ;                /* Open the output file */
  i = 0 ;
  while ( ! feof( fin ) ) {         /* While there is still data, generate */
    StageII( i++, fin, 1, fout, 1 ) ;      /* work for the second stage */
  }
  fclose ( fin ) ;                                /* Close the input file */
  fclose ( fout ) ;                              /* Close the output file */
  return ;
}
```

Figure 5-14 —Source code for the first stage of the three-stage pipeline example, `StageI`.

closes the input and output files and exits normally. `StageI` does not do any computations nor are any of the return values from the second stage used.

The second stage of the pipeline, `StageII` (Figure 5-15), checks if the input file has some data left to process. The output header is written to the output file. The function does some computations (`ComputeStageII`) and then enters a loop calling the `StageIII` function. This loop exits if either the end-of-file is reached for the input file or `WORKBLOCK` calls are made. The latter condition was added to permit investigating the effects of early release of the input file pointer, `sin`. The return value of `StageIII` is ignored. After the loop exits, the comment indicates the location where the early release of the input file pointer can be placed. Currently, the function that releases the file pointer is inserted by hand. Future work will have the function inserted by the precompiler. The `StageII` function then writes to the output file and returns to `StageI`.

```
int StageII ( int N, FILE *sin, int nin, FILE *sout, int nout )
{
  int j, WORKBLOCK = 4; /* A counter and the maximum number of blocks */
  if ( feof( sin ) )  return 1 ;               /* No data, return error */
  PrintHeader( sout ) ;                      /* Print header information */
  ComputeStageII() ;                              /* Do the computations */
  j = 0 ;                                            /* Zero the counter */
  while ( ! feof(sin) && j < WORKBLOCK ) {      /* Generate work for the */
    StageIII( N, j++, sin, nin, sout, nout ) ;              /* third stage */
  }
  /* Early release of sin inserted here */
  PrintTrailer( sout ) ;                    /* Print trailer information */
  return 0 ;
}
```

Figure 5-15 —Source code for the second stage of the three-stage pipeline example, `StageII`.

The third stage of the pipeline, `StageIII` (Figure 5-16), reads a value from the input file. If there is a problem, it returns an error value to `StageII` which currently ignores the value. Again, the location of the early release code for the input file pointer, `tin`, is indicated by a comment. `StageIII` completes its computation by writing to the output file using the file pointer `tout`.

```
int StageIII ( int N, int K, FILE *tin, int nin, FILE *tout, int nout )
{
  int input, status ;          /* Data input variable and status variable */
  if ( feof( tin ) ) return 1 ;                    /* No data, return error */
  status = fscanf( tin, "%d", &input ) ;               /* Read in the data */
  if ( status != 1 ) return 1 ;                      /* There is a problem */
  /* Early release of tin inserted here */
  ComputeStageIII( input ) ;                           /* Compute the data */
  PrintRecord( tout, N, K, input ) ;                   /* Write the output */
  return 0 ;                                   /* Return to the second stage */
}
```

Figure 5-16 —Source code for the third stage of the three-stage pipeline example, `StageIII`.

An alternate method which avoids the end-of-file check of the input file pointer (`sin`) is given for `StageII` (Figure 5-17). This approach has the `StageII` function wait for a non-zero return from `StageIII`. The `StageIII` asset code is written so that if it fails on a read (typically, an end-of-file), it returns a one. This approach fails to achieve added concurrency because of the *future* created in `StageII` on the return variable of `StageIII`.

Figure 5-17 shows an Enterprise technique used to increase computational concurrency. The return variable is converted into a vector and an additional loop, to check on the vector's contents, is added. Aside from being somewhat convoluted and counter-intuitive, this method is less flexible when the I/O behaviour is changed. If the global file semantics are changed to segmented semantics, the extra blocks of work will cause the file pointer to point past the end-of-file when constant segmentation is used.

The segmentation function will have to be carefully constructed when it encounters the end-of-file. Returning an error value (-1) causes the run-time system to attempt to recover from the error. (Currently, the recovery attempt is to abort the application.) Returning a zero causes the run-time system to assume the file segment is of **unknown** size. This approach creates one extra piece of work for `StageIII`. However, this extra work is insignificant since the `StageIII` process does no work as the function immediately returns upon

```
#define WORKBURST 4
int StageII ( int N, FILE *sin, int nin, FILE *sout, int nout )
{
  int j, returnVal[WORKBURST] ;      /* Counter and return variable array */
  if ( feof( sin ) ) return 1 ;                    /* No data, return error */
  PrintHeader( sout ) ;                        /* Print header information */
  ComputeStageII() ;                               /* Do the computations */
  for ( j = 0 ; j < WORKBURST ; j++ ) {      /* Generate a burst of work */
    returnVal[j] = StageIII( N, j, sin, nin, sout, nout ) ;
  }
  for ( j = 0 ; j < WORKBURST ; j++ ) {            /* Consume the futures */
    if ( returnVal[j] == 1 )
      break ;        /* First failure indicates the input file is exhausted */
  }                                  /* End of checking the bursts of work */
  /* Early release of sin inserted here */
  PrintTrailer( sout ) ;                      /* Print trailer information */
  return 0 ;                              /* Return to first stage process */
}
#undef WORKBURST
```

Figure 5-17 — `StageII` asset code modified to check futures.

detecting the end-of-file.

If the asset is configured as *unordered* (see Appendix A for more details) from a computational viewpoint, these *futures* are resolved on an as-received basis instead of on an as-generated basis. At first glance, this looks as if an error in the application is possible, since an out-of-sequence value aborts the work loop prematurely. However, any access to the output file pointer, `sout`, by `StageII` will block until all the outstanding `StageIII` processes have released their claim on the output file.

5.3.4 Extended Pipeline Performance

The experiments presented for the pipeline performance used the same input file. The file consists of 16 pieces of work. The machines used are all Sun4 ELC's with 12 megabytes of memory and a local disk for swap and temporary files. They are connected by a 10Mbps Ethernet network. The sequential time (user time) was 173 seconds which gives a computational granularity of approximately 11 seconds per piece of work. All parallel runs reported are the average of five runs.

A number of combinations are possible for the process interconnection pattern for this example. Recalling the example presented in Chapter 3.4 and Figure 3-6, Figure 5-18 shows the four combinations selected for this experiment.

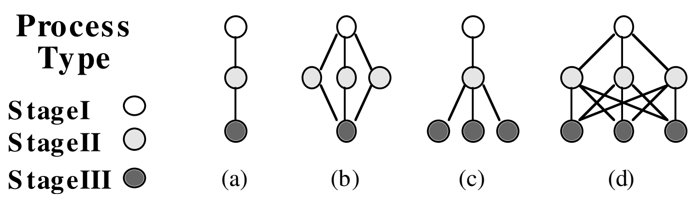

Figure 5-18 — Four computation configurations for three stage pipeline example

The first configuration, Figure 5-18a, is a simple pipeline consisting of three processes. Segmenting the input and output files at all stages will maximize the overall concurrency of the application. However, global file behaviours should not be dismissed. Table 5-4 shows the results of using three combinations of global and segmented I/O behaviours. There are four choices for early release of the input file pointer. The exclamation mark (**!**) indicates that early release was not done. For example, the column labeled **!II&!III** means that both the `StageII` and `StageIII` processes did not release the input file pointer early.

The global input file pointer, `fin`, blocks the first stage, `StageI`, when the loop checks for EOF of the input file. Appropriately, only one `StageII` process can be active. This same check on the input file pointer, `sin`, blocks the `StageII` function when it checks for the EOF. Again, since only one `StageIII` process is active, this is acceptable. Unless the input file is released early, the `StageII` and `StageI` processes are blocked waiting for access after generating one piece of work. However, since there is only one process per

File pointer names				Early Release			
sin	**sout**	**tin**	**tout**	**!II&!III**	**II&III**	**II&!III**	**!II&III**
Meeting	Meeting	Meeting	Meeting	193	193	193	193
Meeting	Newspaper	Meeting	Newspaper	188	188	188	188
Newspaper	Newspaper	Newspaper	Newspaper	185	184	184	184

Table 5-4 — Elapsed times for different combinations of parallel I/O behaviours and early release using the computational pattern shown in Figure 5-18a. Sequential user time is 173 seconds.

stage, early release does not show any benefit. The segmented approach does not show superior performance over global behaviours.

The output file pointer does not benefit from early release as the function returns immediately after the last access with it. If the output file pointer is defined as having global behaviour, only one process can be active in the file at a time. Since all the computations are done, the output should be completed quickly to allow the next waiting process to access the file. Segmenting the output file is somewhat difficult as the record size is not known until after the record is written. Using **unknown** segment sizes is appropriate for this application.

None of the combinations showed performance better than the sequential version. The lower the demand for synchronization (changing from global to segmented parallel behaviours), the better the performance. PI/OT limits the concurrency between stages by imposing barriers at each stage which now cause a process to wait for its child process to return I/O information. Effectively, only one process at a time is allowed to progress. The cost of messages further reduces the application performance.

The second process configuration, Figure 5-18b, attempts to improve the performance. The configuration replicates the second stage three times and leaves the third stage as a single process. The changes to the computational parallelism are made to the graph file and leave the I/O parallelism unchanged. No recompilation is necessary. The `StageIII` process performs the bulk of the computations (number of invocations) and is rapidly overwhelmed with requests for work from the three `StageII` processes. The data in Table 5-5 show that the parallel performance is comparable to the previous configuration (Figure 5-18a) but uses more processors. Even with the early release of the input file pointer, `sin`, there is not much concurrency available since there is only the one `StageIII` process. Using a global behaviour for the input file pointer with early release and segmented behaviour for the output file pointer yields the best performance of this configuration. Segmenting the input file increases the concurrency but again, the `StageII` processes overwhelm the single `StageIII` process. Unless the cost of computing the second stage is significantly more than the third stage, this configuration of processes is not that successful.

File Pointer Variables				**Early Release**			
sin	**sout**	**tin**	**tout**	**!II&!III**	**II&III**	**II&!III**	**!II&III**
Meeting	Meeting	Meeting	Meeting	193	172	192	192
Meeting	Newspaper	Meeting	Newspaper	188	166	187	188
Newspaper	Newspaper	Newspaper	Newspaper	168	168	167	167

Table 5-5 — Elapsed times for different combinations of parallel I/O behaviours and early release using the computational pattern shown in Figure 5-18b. Sequential user time is 173 seconds.

The third process configuration, Figure 5-18c, has the last stage replicated three times. Since the bulk of the computations are done in the last stage, replicating the last stage should yield a better result than the previous two attempts. Using global behaviours, the one `StageII` process would wait for a `StageIII` process to return control in order to proceed. The mixture of global input and segmented output file pointers would not show a significant improvement since each of the `StageIII` processes would wait for access to the global input pointer. However, using early release of the input file pointer for this stage in the pipeline, the three `StageIII` processes will become fully utilized. Performance should approach the performance of the fully segmented file pointer test.

The results of using three different PI/OT template combinations and invoking early release of the input file pointer, shown in Table 5-6, indicate that early release of the input file pointer in the `StageIII` processes does permit more overlapping concurrency. The pure segmented behaviour fills the pipeline and shows the best performance. The mixture of global and segmented file pointers shows only slight improvement over the global behaviour. The effect of adding early release to the last stage significantly improves the application's performance over its sequential performance.

File Pointer Variables				Early Release			
sin	**sout**	**tin**	**tout**	**!II&!III**	**II&III**	**II&!III**	**!S&III**
Meeting	Meeting	Meeting	Meeting	194	111	194	112
Meeting	Newspaper	Meeting	Newspaper	189	109	188	109
Newspaper	Newspaper	Newspaper	Newspaper	105	104	105	105

Table 5-6 — Elapsed times for different combinations of parallel I/O behaviours and early release using the computational pattern shown in Figure 5-18c. Sequential user time is 173 seconds.

The fourth process configuration, Figure 5-18d, has both the second and third stages replicated three times. Using only global behaviours with early release should show better results since more of the second stage processes would be concurrently utilized. With the second and third stages replicated, the early release of the input file pointer is beneficial only if either the third stage or both the second and third stages are involved. This is obvious since early release of the second stage relies on the third stage releasing the input file pointer first. The purely segmented approach would show no effect using early release since the input file segment is not modified. The data in Table 5-7 confirms these predictions.

File Pointer Variables				Early Release			
sin	**sout**	**tin**	**tout**	**!II&!III**	**II&III**	**II&!III**	**!II&III**
Using a replication factor of three for both `StageII` and `StageIII` Processes							
Meeting	Meeting	Meeting	Meeting	193	90	192	110
Meeting	Newspaper	Meeting	Newspaper	189	73	189	108
Newspaper	Newspaper	Newspaper	Newspaper	67	67	67	67
Using a replication factor of three `StageII` and six for `StageIII` Processes							
Meeting	Meeting	Meeting	Meeting	193	62	192	83
Meeting	Newspaper	Meeting	Newspaper	189	44	188	71
Newspaper	Newspaper	Newspaper	Newspaper	39	37	37	38

Table 5-7 — Elapsed times for different combinations of parallel I/O behaviours and early release using the computational pattern shown in Figure 5-18d. Sequential user time is 173 seconds.

The fifth configuration is similar to the fourth configuration except that the replication factor for the third stage is increased to six for a total of eleven processes. When the replication factor of the last stage was increased to six, the segmented performance continued to improve[3]. As well, the mixed behaviour version using early release continued to show an improvement as more of the `StageIII` processes were utilized.

5.3.5 Useability and Composability Summary

The PI/OT model offers many choices for parallelizing the I/O. The two example applications, while containing relatively simple computational parallelism, show how more

[3] When repeating this experiment on the heterogeneous network used in the first two experiments (Section 5.1 and Section 5.2), increasing the replication factor of the last stage actually degraded performance. This happened since it is more likely that a slower processor would be selected by the computational manager.

complex I/O patterns are easily created by composing these simple parallel behaviours. The ease of developing the parallel applications without any dependencies on specialized library functions or by explicitly encoding the parallelism into the application is a positive feature of this approach to parallel I/O. These two examples also show that parallelizing I/O for more complex applications does not readily show the speed up that the computational parallelism suggests is possible. Granularity of the computations must be increased to balance the cost of the synchronization and coordination of the parallel I/O model. The current deadlock prevention mechanism does not permit as much flexibility as desired with the read and write attributes. The more I/O pointers involved in a transaction, the less flexibility is shown. More work is needed to improve the deadlock prevention algorithm to gain tangible proof of the benefits of three levels of ordering.

5.4 Dynamic Segmentation

Strided interfaces or pre-defining access structures for regular data structures appears to be the current approach to segmenting a file. This is acceptable for regular data structures like dense matrices. Irregular sized data structures are not as easily manipulated by a predefined or regular segmentation approach and consequently, show poor parallel performance.

An irregular data structure is typically represented in a file as a header element indicating either the size or number of elements — a description of what follows. The actual elements follow the header element. This two step implementation is often nested. While performing two read or write operations works well with sequential applications, a two stage read or write technique is difficult to manage for parallel applications because the concurrency introduces synchronization problems. Another approach to interpreting an irregular data record is to use a special character indicating the end of record. For example, the segmentation function in Figure 4-14 uses the knowledge that every third line feed character indicates the end of a data record to segment a file.

Sequentially, a file is treated as a stream of bytes. No structure is imposed on the disk file. In parallel applications, this lack of structure causes problems. Segmenting a file not based on meta-information regarding the size and composition of data blocks could be inefficient. Sequentially, this information is found by performing two reads or double writing a complex object. Double writing means that the object is written to a temporary buffer (disk or memory), the record length determined, and the record is copied to disk along with the length. When importing a sequential file into a parallel I/O system, a user should be given a chance to cache segmentation information for a specific data file. This could be done either by creating a meta-file or by modifying the contents of the actual data file. This last step is not done casually since it effects the user's code.

During the parallel computation, the meta-information about the file structure is used to segment the file. Export of the file from the parallel file system back to the sequential file system uses this meta-information to reassemble the parallel file. Both the import and export steps should be taken into account when determining overall processing time for the files. In effect, the application reads the file twice — once for the meta-information and once for the actual data. Since there is little chance of avoiding this double read, a segmentation function that does this as the application progresses through the file is proposed.

Earlier in Section 5.1, a fine-grained I/O application was used to compare the performance of PI/OT and PIOUS. That particular comparison used a constant segmentation factor. This section looks at the cost of using a dynamic segmentation function.

5.4.1 Segmentation Functions

Three segmentation functions were tested. These three form a spectrum of segmentation functions. They range from a complete reading of a data record to a single small (4 byte) read to establish the size of the record to no read (a constant). From this spectrum, the effect of the approach to segmenting a file is studied.

The first approach is a complete read of the entire record (Figure 5-19). Instead of using the version shown in Figure 5-5 which seeks over the elements of a subcomponent, the function reads the size of a subcomponent. The function then reads in the elements to advance the file pointer to the start of the next subcomponent. Since the size of the data is known or computable, reading into a buffer is acceptable. Of the three segmentation functions tested, this function should have the largest impact on performance. The segmentation function effectively reads the entire segment using a number of small I/O operations.

```
#define IC ( sizeof(int) + sizeof(char) )
#define I sizeof(int)
unsigned long ReadSegmentation ( FILE * fp, int min, int max, int current )
{
  unsigned long offset ;                               /* Extent of this record */
  int CD, C, D, E, i ;                 /* Record header variables and a counter */
  char buffer[ 4096 ] ;                /* The maximum size of a record on disk*/
  i = fread( &CD, I, 1, fp ) ;            /* How many CEDE objects are there */
  if ( i != 1 )                       /* End of file or file error, return error */
    return (unsigned long)-1 ;
  offset = I ;                          /* The record includes the size of CD */
  for ( i = 0 ; i < CD ; i++ ) {          /* Loop reading the CEDE records */
    fread( &C, I, 1, fp ) ;                        /* Elements in this C record */
    fread( buffer, sizeof(char), C * IC, fp ) ;          /* Read C data block */
    offset += I + C * IC ;                         /* Increment size of record */
    fread( &E, I, 1, fp ) ;                        /* Elements in this E record */
    fread( buffer, sizeof(char), E * IC, fp) ;           /* Read E data block */
    offset += I + E * IC ;                         /* Increment size of record */
    fread( &D, I, 1, fp ) ;                        /* Elements in this D record */
    fread( buffer, sizeof(char), D * IC, fp ) ;          /* Read D data block */
    offset += I + D * IC ;                         /* Increment size of record */
    fread( &E, I, 1, fp ) ;                        /* Elements in this E record */
    fread( buffer, sizeof(char), E * IC, fp ) ;          /* Read E data block */
    offset += I + E * IC ;                         /* Increment size of record */
  }                                  /* End of loop reading in the CEDE records */
  fread( &E, I, 1, fp ) ;                          /* Elements in this E record */
  offset += I + E * IC ;                           /* Increment size of record */
  return offset ;                        /* Return the size of the Child record */
}
#undef IC
#undef I
```

Figure 5-19 —Segmentation function for fine-grained example that reads the entire record.

The second function, `EmbeddedSegmentation` (Figure 5-20), requires modifications to the sequential and parallel applications. The internal structure of the file must be modified

```
unsigned long EmbeddedSegmentation ( FILE * fp, int min, int max, int current )
{
  int size, i ;
  i = fread( &size, sizeof(int), 1, fp ) ;        /* How big is this record */
  if ( i != 1 )                     /* End of file or file error, return error */
    return (unsigned long)-1 ;
  /* Return the size (in bytes) of this record plus the header size */
  return (unsigned long)(size + sizeof(int);
}
```

Figure 5-20 —Segmentation function for fine-grained example that has the size of the record embedded into the data file.

to include a header before each `Child` data block indicating how many bytes the block contains. The function reads in the value and returns the offset. This intrusive approach should have a smaller impact on performance than that of the full read version because there is only one small read operation.

The third function, `ConstantSegmentation` (Figure 5-21), is intended to have the least impact on performance. It simply returns a constant value with no access to the disk. However, this constant size trades knowledge for flexibility. If the records are smaller than this constant size, holes will exist in the file. If the data records are larger than the stated size, data is lost. As well, the program is corrupted due to reading past segment boundaries or the file is corrupted by writing past segment boundaries.

```
unsigned long ConstantSegmentation ( FILE * fp, int min, int max, int current )
{
  /* Return the size (in bytes) of this record */
  return (unsigned long) 352108;
}
```

Figure 5-21 — Constant segmentation function for fine-grained I/O example.

5.4.2 Dynamic Segmentation Performance

The two read segmentation functions and the constant value function were used to examine the effect of increasing computational granularity against increasing the replication factor. The application was run using four different computational granularities consisting of approximately 0, 10, 37 and 147 seconds per data record (Table 5-8).

This granularity is based on the computational part of the application, not the cost of segmenting the record. Each computational granularity was tested using four different replication factors of 2, 5, 10, and 15 `Child` processes for each segmentation function. All parallel runs reported are the average of five runs. The data indicates that until a certain

Replication Factor	CPU granularities (seconds)			
	0	10	37	147
Sequential	16	479	1853	7339
Full Read Segmentation				
2	239	512	1361	4875
5	168	251	535	1824
10	137	187	324	1062
15	138	148	255	679
Embedded Read Segmentation				
2	210	505	1361	4870
5	127	225	534	1825
10	119	160	332	1069
15	118	147	266	697
Constant Segmentation				
2	208	505	1372	4869
5	123	220	531	1824
10	115	155	330	1067
15	115	142	264	695

Table 5-8 — Elapsed time (seconds) using three different segmentation functions, four replication factors for the `Child` process, and four computational granularities for the fine-grained I/O example.

threshold of computational granularity is reached, the application is better off being run sequentially. There appears to be little difference in performance to determine which segmentation function is best to use.

A maximum of sixteen processors are used in this set of experiments. Fourteen are Sun4 ELCs while the other two were slower Sun4 IPCs. When using the IPCs, any administrative processes (e.g. the Enterprise `root` process[4]) or a process that did not have a large CPU requirement were placed on the slower machines. Where possible, the fastest processors were used first. All processors have a local disk for swap and temporary files and are connected by a 10Mbps Ethernet network.

Figure 5-22 (a), (b), and (c) shows the performance of the three segmentation functions with increasing replication factors for the `Child` processes. These figures show that there

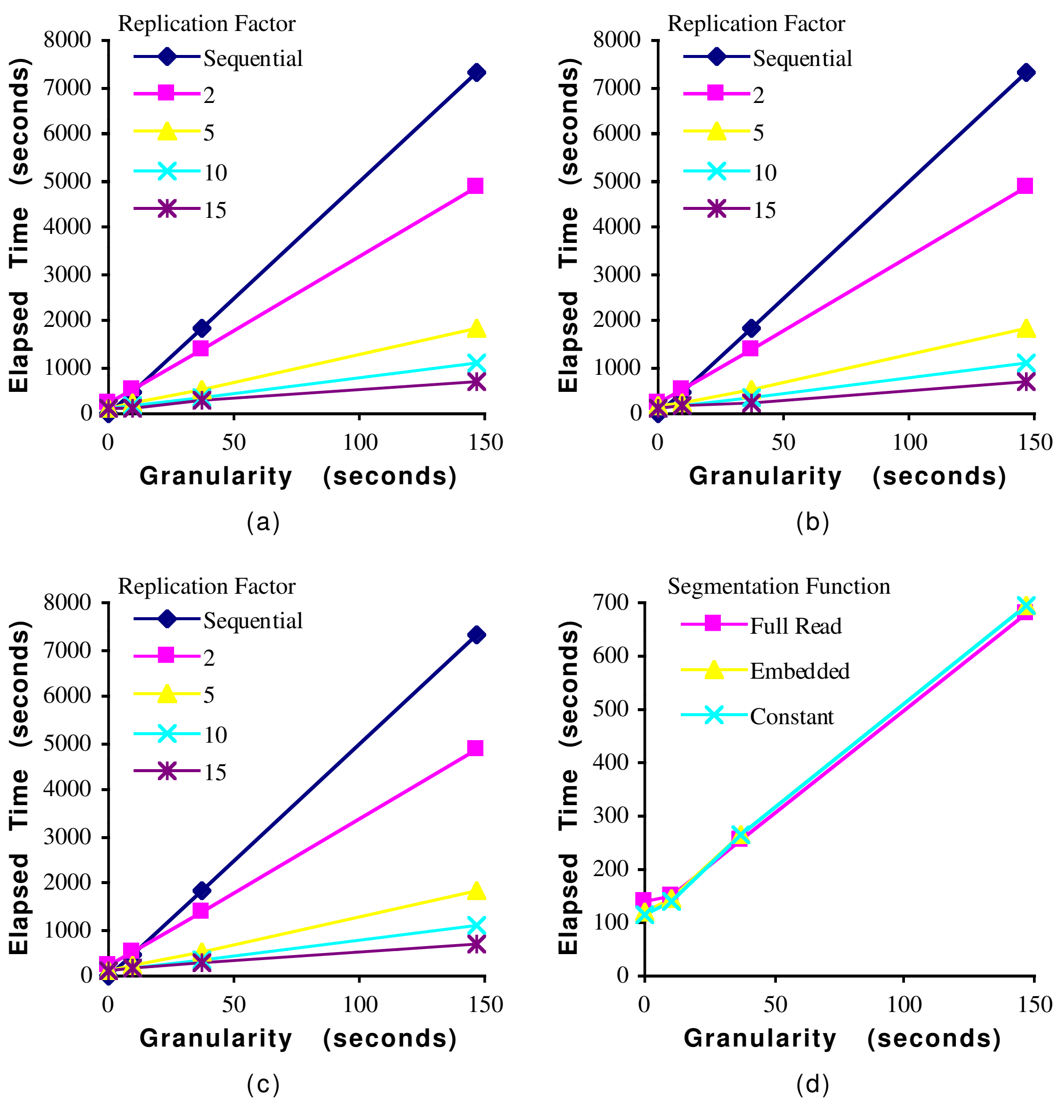

Figure 5-22 — Elapsed time versus computational granularity using constant (a), full read (b), and embedded read (c) segmentation for fine-grained I/O example at four replication levels. Elapsed time versus computation granularity of the three segmentation functions using a replication factor of fifteen (d).

[4] See Appendix A for more details.

are diminishing returns by increasing the replication factor. This diminishment can be attributed to the Parent process becoming a bottleneck for distribution of the input file segments and integrating the output file segments. The computational interactions limit the scalability of this implementation of PI/OT. The difference between the three different segmentation functions using a replication factor of 15 is seen in Figure 5-22(d). Unless the computational granularity is low, there is no significant difference between the three approaches to segmentation.

At low granularity or replication factor, the constant function and the embedded read function show definite performance improvements over the full read segmentation function. The cost of fully reading the record twice is noticeable but small. However, once there was sufficient computational granularity (10 seconds), all the segmentation functions gave similar performance. The full read version gave slightly better performance with the higher granularities and replication factors.

The performance improvement can be attributed to the overlapping I/O and computations. As well, the I/O has been prefetched and is still likely to be either in the disk cache or resident in memory at the file server. Clearly, the costs of segmentation are hidden when there is sufficient computational granularity. For this application, the Parent process continues to segment the file after all the Child processes are busy. The overlap of the I/O and computations is beneficial.

5.4.3 Dynamic Segmentation Summary

Three segmentation functions were examined for their relative performance in a parallel application. As the granularity of the computation rose, the choice of a segmentation function became moot. Clearly, the least interference with the original data file is most desirable.

The performance of the simple parallel configuration is acceptable (in that a speedup is seen). The next section looks at a more complicated computational pattern along with an increase in the complexity of the parallel I/O requirements. The intent is to examine whether improved performance can be extracted for the application and to determine the costs of creating the application.

5.5 Complex I/O Patterns

If there is sufficient computational granularity, experimentation with the configuration of a parallel application may yield better performance. The fine-grained example used in Section 5.1 and 5.4 was modified to change the computational parallelism in order to seek better performance on the two computational platforms.

The Child data record (Section 5.1.1) consists of **n(CEDE)** data blocks followed by a trailing **E** data block. With increasing computational granularity, computing a subset of the **CEDE** blocks could be parallelized with the potential for an increase in performance. However, the I/O requirements have now been changed. What modifications are necessary to implement these changes?

A new function, CEDE, was developed to run in parallel (Figure 5-23). This function assumes that what is left in the input file is a collection of CEDE records. This approach takes advantage of the PI/OT segmented I/O semantics in which the end-of-file is the same as end-of-segment for the newspaper template. An alternate approach has the actual number of records in the segment block passed as a parameter to the function. However, in either case, the parallel I/O implementation must leave the input file pointer located at the start of the trailing **E** record for the Child process after all the calls to CEDE.

The computational parallelism is straightforward to modify. The computation becomes a three-stage pipeline consisting of Parent as the first stage, Child as the second and CEDE

```
int CEDE( FILE *cdin, int nin, FILE *cdout, int nout ) {
  if ( cdin == NULL ) return 1 ;                    /* No file is defined! */
  while ( ! feof(cdin) ) {/* Assume the file contains only CEDE blocks */
    ComputeC( cdin, cdout ) ;                          /* Compute the C record */
    ComputeE( cdin, cdout ) ;                          /* Compute the E record */
    ComputeD( cdin, cdout ) ;                          /* Compute the D record */
    ComputeE( cdin, cdout ) ;                          /* Compute the E record */
  }
  fflush( cdout ) ;                        /* Ensure output has gone to disk */
  return 0 ;                                   /* Return to Child function */
}
```

Figure 5-23 —Source code for the CEDE function for the more complex I/O example based on the fine-grained I/O example.

as the third. The second and third stages are replicated. The original computational configuration is seen in Figure 5-24a while Figure 5-24b shows the new configuration.

For performance comparison, the number of actual processes is kept the same. A `Child` process still receives an input file segment from the `Parent` process. `Child` is now responsible for distributing the sub-segments of its segment to the CEDE function. The `Child` process must also collect the output from these client processes and merge the output before returning its output segment to `Parent`.

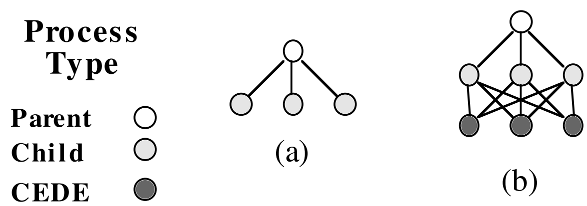

Figure 5-24 — The computational parallelism for original (a) and more complex (b) version of the fine-grained I/O example.

The `Child` function required some modification (Figure 5-25) in order to implement this new parallel computational behaviour. The loop counter where the **CEDE** blocks were computed was changed to increment by a specified number of blocks. Inside the loop, the number of **CEDE** elements per block was monitored to ensure that only **CEDE** blocks were distributed. After the loop finished, the trailing **E** record was computed. The `Child` process then waited until all the outstanding CEDE work requests were completed and returned their output before returning to the `Parent` process.

The only PI/OT modification is that an additional segmentation function must be created to segment the **CEDE** record blocks. The next section documents the creation of this additional segmentation function.

5.5.1 An Additional Segmentation Function

A new segmentation function (Figure 5-26) was created for the new computational stage, CEDE. One problem arose when developing the sequential code. If a constant block of **CEDE** records is used, there is a good possibility that the last block will contain more than just **CEDE** records. For example, suppose there were 101 **CEDE** records in the `Child` segment and the blocking factor was ten. The last file segment should only contain one **CEDE** record but it will contain additional information that the CEDE function will misinterpret and corrupt the rest of the calculations. Since the interface between the callback function and run-time system is already defined, an alternate approach is needed.

```
#define StartingCEDEBlock 10
int  CEDE_BLOCKSIZE  =  StartingCEDEBlock  ;/* Used by segmenation function */

int Child( FILE *bin, int nin, FILE *bout, int nout ) {
  int Cdelem, I;              /* Number of records to process and a counter */
  char type='B';                                          /* Record type */
  i = fread( &CDelem, sizeof(int), 1, bin ) ; /* Number of CEDE records */
  if ( i == 0 ) return 1 ;                /* Read failed, nothing to do */
  if ( ! feof( bin ) ) {                             /* Not end of record */
    OutputBHeader( bout, CDelem );        /* Output the B record header */
    CEDE_BLOCKSIZE = StartingCEDEBlock ;        /* Compute a CEDE block */
    for ( i = 0; i < CDelem; i += CEDE_BLOCKSIZE ) {/* Distribute blocks */
      /* Don't exceed the number of records ! */
      if ( i + CEDE_BLOCKSIZE > CDelem ) CEDE_BLOCKSIZE = CDelem - i ;
      CEDE( bin, 1, bout, 1 );                  /* Compute the CEDE block */
    }
    ComputeE( bin, bout );               /* Compute the trailing E record */
  }
  fflush( bout ) ;                      /* Ensure output has gone to disk */
  return status ;             /* Return status of this function to Parent */
}
```

Figure 5-25 —Modified source code for the `Child` function reflecting the changes necessary for the more complex fine-grained I/O example.

One solution is to use a global variable. The `Child` process knows the number of **CEDE** blocks prior to invoking the `CEDE` function. It was a simple matter to insert a test in the `Child` distribution loop to test and, if necessary, limit the number of **CEDE** blocks distributed. This global variable could be declared `static` and effectively shielded from

```
extern int CEDE_BLOCKSIZE ;            /* How many CEDE blocks to read */
#define IC ( sizeof(int) + sizeof(char) )
#define I sizeof(int)
unsigned long segCEDE( FILE *fp, int curr, int min, int max ) {
  unsigned long offset = 0 ;
  int i, C, D, E ;
  char buffer[4096] ;                /* Maximum size of a record on disk */
  for ( i = 0 ; i < CEDE_BLOCKSIZE ; i++ ) { /* Determine the block size */
    fread( &C, I, 1, fp ) ;                  /* Elements in this C record */
    fread( buffer, sizeof(char), C * IC, fp ) ;     /* Skip over C record */
    offset += I + C * IC ;                  /* Increment the size counter */
    fread( &E, I, 1, fp ) ;                  /* Elements in this E record */
    fread( buffer, sizeof(char), E * IC , fp ) ;    /* Skip over E record */
    offset += I + E * IC ;                  /* Increment the size counter */
    fread( &D, I, 1, fp ) ;                  /* Elements in this D record */
    fread( buffer, sizeof(char), D * IC, fp ) ;     /* Skip over D record */
    offset += I + D * IC ;                  /* Increment the size counter */
    fread( &E, I, 1, fp ) ;                  /* Elements in this E record */
    fread( buffer, sizeof(char), E * IC, fp ) ;     /* Skip over E record */
    offset += I + E * IC ;                  /* Increment the size counter */
  }
  return offset ;             /* Return the size of this CEDE record block */
}
#undef IC
#undef I
```

Figure 5-26 — Segmentation function for `CEDE` parallel I/O requirements.

the rest of the code if the asset code and the segmentation function share the same file.

5.5.2 Complex I/O Performance

The number of **CEDE** blocks per segment sent to a remote `CEDE` function was set to a default value of ten. Three replication combinations using ten processes were tested. Two different computational platforms were used. The same operating system was used on both platforms — SunOS 4.1.4. All the processors communicate with each other using a 10Mbps Ethernet connection.

The first platform is heterogeneous in processor capacity. It is similar to the configuration used in the first two experiments (Section 5.1 and Section 5.2) except that the Sparc 4 (SS4) and one of the Sun ELCs is now replaced by a Sparc 10 (SS10) dual processor unit and a Sun SLC. The SS10 is about 4.5 times faster than the SLC. The other processor ratios are found in Section 5.1.5. The configuration consisted of the following processors (and memory): one SS10 (96 megabytes), two Sun4 Classics (32 megabytes), five Sun ELCs (32 megabytes), and five Sun SLCs (16 megabytes). All processors have local disk for swap and temporary file space. The second configuration is the homogeneous platform that was used for the third (Section 5.3) and fourth (Section 5.4) experiments, and consists of thirteen Sun ELCs with 12 megabytes of memory and local disk for swap and temporary file space.

This problem contained fifty `Child` records with each `Child` record containing one hundred **CEDE** records. As the number of `Child` processes increases, the `CEDE` processes become the bottleneck. For the data file used, each time a `Child` process receives a work request from `Parent`, `Child` generates ten `CEDE` calls. For these experiments, the sequential time needed to process one `Child` record was set to be about 85 seconds on the fastest processor that runs a `Child` process in the heterogeneous case. That same setting corresponds to 147 seconds with the homogeneous process cluster. The average computational cost for a `CEDE` call is approximately 9 (heterogeneous) or 15 (homogeneous) seconds. With eight `CEDE` processes, the twenty requests are quickly handled. As the number of work requests increases, the reduced number of `CEDE` processes, which do the majority of the computations, inhibits performance on the homogeneous processor cluster. However, using the heterogeneous process cluster, the slower processors were specifically selected to run the `Child` processes, leaving the `CEDE` processes to the faster processors. Consequently, there should a noticeable improvement to the performance of the application.

The `Child` file fragment was further segmented for each block of `CEDE` records. There is little impact on the network for I/O activity as all the `Child` processes segment their local copy of the larger file fragment. Similarly, the `Child` processes reassemble the output of the `CEDE` processes before returning their larger segment to the `Parent` process. As the `Child` and `CEDE` processes do not know in advance the size of their respective output record, an **unknown** segment size was used for both process types.

Given the same number of processes, Table 5-9 shows that the performance is better

Replication		Elapsed Time (seconds)		CEDE
Child	CEDE	Heterogeneous	Homogeneous	Requests
10	0	1538	1062	0
2	8	1448	1023	20
3	7	1516	1158	30
4	6	1630	1344	40

Table 5-9 — Elapsed time in seconds for a more complex computation on a heterogeneous and a homogeneous network of workstations. A total of ten processes are allocated to execute the `Child` and `CEDE` functions.

than in the simpler pipeline example. The shaded row corresponds to the elapsed time for the simpler two-stage pipeline approach. There is about a six percent increase in the performance on the heterogeneous processor cluster using the more complicated parallel design. This performance improvement deteriorates as the number of `Child` processes increased. This is due to the inability of the faster processors to keep up to the increased number of work requests from the `Child` processes. The homogeneous computational platform showed very little performance gain and, as the number of `Child` processes increased, performance degraded quickly.

5.5.3 Complex I/O Summary

The benefits of a template approach over a hand-coded library approach are highlighted in this experiment. With a minor change to the source code, the performance is incrementally improved for the heterogeneous network. Adapting to the change involved a modification to the computational parallelism and corresponding adaptation to the I/O parallelism. The PI/OT and Enterprise modifications took about an hour to design and implement. The application needed to be recompiled by Enterprise to reflect the changes in the computational parallelism (creating the new wrapper function for `CEDE`). Creating this same application by integrating and embedding both the computational and I/O parallelism into the code, for example using PVM and PIOUS, would take considerably more time and would be prone to errors. Clearly, while PI/OT does not parallelize the application, this model of I/O provides the abstractions necessary to allow the user to specify **what** is needed while leaving **how** to implement requirements to the run-time system.

5.6 Chapter Summary

Five sets of experiments examined the performance and useability of the PI/OT model and implementation. The first two experiments compared the performance aspect of the Enterprise implementation of PI/OT and a more low-level parallel file system, PIOUS. Both experiments had the same computational parallelism — a simple parent-child behaviour. Enterprise and PIOUS both use PVM as the underlying communication system. The difference is that Enterprise uses computational templates to express the parallelism while the PIOUS version relies on the user to hand-code the necessary parallel communications to develop the parent-child behaviour. The computational behaviour was straightforward to implement in either system. However, the parallel I/O requirements were quite different.

The first experiment (Section 5.1) has fine-grained I/O requirements spread throughout the application. The first experiment had one input and one output file, each segmented. Each segment of work consisted of many small I/O calls involving from four to several hundred bytes. Only one Enterprise version gave acceptable performance.

The hand-coded approach using PIOUS permitted seven different implementions of the same application. The user could use the PIOUS system calls to treat the files as either segmented or globally shared, and cache segments to a local disk or not. The local processes used buffered or standard I/O streams, or low-level I/O calls on these local cached copies. Each of the PIOUS versions developed took several hours to code and debug. The child process needed no modifications to the original source code for the child except in one case where all I/O was replaced with the PIOUS I/O calls.

The second experiment (Section 5.2) had a few coarse-grained I/O operations spread over three file pointers. The application did not provide the same variety of choices as did the fine-grained I/O when using the low-level parallel file system. In fact, there was only one way of implementing it in each parallel I/O system to get acceptable performance.

The third experiment (Section 5.3) examined the useability and composability of the PI/OT parallel I/O model. Changing the external parallel I/O requirements did not require a recompilation (or re-writing) of the sequential source code. Early release of file pointers shows the improved concurrency for a given application.

Early release is only one part of the potential of using static analysis to improve I/O performance. The compiler can determine the scope of a file pointer's use in a function. The early release mechanism can be inserted when the file pointer is no longer needed. Static analysis can also determine if it can safely rearrange code to cluster the I/O statements to improve the effect of early release. This analysis can also be used to provide hints to the run-time system indicating that, for a given transaction type, deadlock prevention is not necessary. However, static analysis can only provide hints. Since the parallel I/O behaviour can be changed without requiring the recompilation of the application, the run-time system must make the final decision about deadlock prevention.

The fourth experiment (Section 5.4) examined the performance of three different segmentation functions, as proposed for the dynamic segmentation contribution of PI/OT. The performance of segmentation functions is not as much of a concern when there is some computational granularity. The exact point when the segmentation function has a significant impact on the performance depends on the size of the computational granularity, the amount of I/O needed, and the complexity of calculating the size of the extent.

The fifth experiment (Section 5.5) revisited the first application, the fine-grained I/O. The lessons learned in the first four experiments were applied to see if the earlier performance could be improved. The application was modified for a more complex computational pattern and a different parallel I/O behaviour. This resulted in a six per cent increase when working with a heterogeneous mix of processors. This same pattern showed little or no gain when running the same application with a homogeneous processor mix. Modifications were needed to the computational parallelism but the only additional code required for the new parallel I/O behaviour was a new segmentation function. The functions using I/O did not need any modifications for the new parallel I/O behaviour. The advantages of a templated approach to the computations and I/O parallelism are seen in the flexibility to experiment and test to seek the best performance for a given computational platform.

Chapter 6

6. Conclusions

The feasibility of a top-down approach to parallelizing the I/O components of an application has been presented. The current approaches are bottom-up with specialized libraries of functions differentiating between parallel and sequential streams. Synchronization and coordination between different I/O streams are left as the responsibility of the user. Transaction support is rudimentary. All the parallelism is explicitly inserted into the application code. If either the computational or I/O parallelism is changed, a new version of the application must be created. PI/OT eliminates these problems by using I/O templates and cooperating with the computational templates. However, given the desire for high performance in parallel applications, the above advantages can easily be negated if the run-time performance of PI/OT is poor. While the current implementation is not complete, the results reported here validate the claim that the performance is acceptable.

In fact, there are two claims made in this thesis. The first claim (Chapter 3 and 4) is that parallel I/O behaviours in an application can be specified separately from the source code. The work presented here extends the template approach which is shown to be beneficial for computational parallelism by Szafron and Schaeffer [73]. The user code is significantly reduced in comparison to the hand-coded versions and the application is completed much sooner since the templates are correctly implemented for the selected parallel behaviour. There are no new parallel I/O function libraries to learn and no language extensions. All I/O is performed using the familiar standard stream interface. The separation of the parallel I/O behaviours from the source code, coupled with a corresponding separation of parallel computation behaviours, can be combined to produce a more responsive and "better" tuned application. This corresponds to the idea that the user specifies **what** parallel behaviours should be used while the parallel programming system determines **how** to implement the behaviours.

The second claim (developed in Chapter 5) is that the software engineering advantages of such an approach do not necessarily incur a significant loss of performance. This claim agrees with the conclusions of Szafron and Schaeffer that only a small performance penalty is paid for using computational templates.

A number of extensions and future research possibilities arise from this work. Section 6.1 documents some of the compile-time and run-time extensions that would lead to further improvements. Section 6.2 lists the contributions of this dissertation. Section 6.3 summarizes the work presented in this document.

6.1 Extensions and Future Research

Not all of the lessons learned from developing this PI/OT prototype are integrated into the current solution. Rather than adding the missing portions (the **report** template, writable **photocopies**, a more intelligent deadlock prevention mechanism, and support for atomic I/O statements) in an ad-hoc fashion, the current implementation would benefit from a complete rewrite.

The Enterprise PPS needs modifications to the precompiler and run-time system to fully support the different I/O templates. Primarily, this involves providing hints for deadlock prevention and prefetching. Enterprise is intended to be a complete parallel programming environment. The Enterprise graphical user interface should be modified to support graphical PI/OT templates. Currently, the graph file is modified by hand. PI/OT generates events about parallel I/O operations which are intended for use by the Enterprise parallel debugger and performance monitor. Both of these components need modification in order to use this information.

A second area of future work is to implement PI/OT within another parallel programming system. The decision to use Enterprise for the prototype was easy to make. The Enterprise source code and, more importantly, the people who developed Enterprise were available for consultation when it came time to implement PI/OT. The I/O model presented here is independent of the implementation platform. However, proof of this independence would be demonstrated by a successful implementation in another parallel programming system. Independence could also be validated by implementing PI/OT using a specialized parallel I/O library. This top-down approach to parallel I/O should be able to take advantage of the optimizations offered by these tools.

As well, Enterprise only supports certain computational models. Mesh and peer-to-peer models, for example, are not supported. The reason for this is that computational deadlock is difficult to detect or prevent for such templates. Implementing PI/OT in a PPS that allows the user to develop such a computational application would test the current deadlock prevention mechanism. The work done so far has identified the minimal requirements for a parallel computational system needed to implement PI/OT.

All this work has been done on a network of workstations (NOW), but shared memory processors (SMP) are a popular competitive alternative. As well, distributed shared memory systems (DSM) can provide adequate performance if the network provides sufficient bandwidth and message latency for a particular application. Can shared memory and parallel I/O co-exist? Consider a file pointer that is placed in shared memory. Unless there is operating system support for such a strategy, problems will develop. In particular, if distributed shared memory is used, a file pointer created by one processor and used on a different processor is not likely to work. In addition, shared memory would also require the memory stream (`sprintf` and `sscanf`) functions to be made parallel-aware. I/O templates would work with the shared memory model and the computational parallelism to provide an appropriate solution. Therefore, a PPS that supports a shared memory model would need compiler and run-time modifications to identify and support shared parallel I/O objects.

The five templates presented here should not be considered complete. Future work is needed to examine whether additional templates or attributes are appropriate and to determine their relationships. For example, the **log** template can be treated as a specialization of the **meeting** template while the **report** template can be considered a generalization of the **newspaper** template. However, these two specialized templates are useful to specify common parallel behaviours.

An important area for future work is the development of a more sophisticated deadlock prevention mechanism. This will require compiler and run-time support. The conservative approach of the current implementation restricts concurrency and the expressiveness of the attributes. Section 6.1.1 addresses this problem in more detail.

The rest of this section describes in more detail where static and run-time extensions should be beneficial. Section 6.1.1 looks at extensions to the deadlock prevention with the PI/OT model. Section 6.1.2 discusses some of the issues that static analysis should address to integrate I/O and computation parallelism. Section 6.1.3 discusses some extensions of the run-time support. Section 6.1.4 summarizes the future research directions of the work presented in this dissertation.

6.1.1 Deadlock Prevention

As discussed in Chapters 3, 4, and 5, deadlock prevention is needed. Deadlock detection after the fact is not appropriate since resolution will likely involve a rollback of computations and I/O operations. This is difficult and costly to do correctly in the general case (if it can be done at all). Deadlock prevention does not require rollback. The price of this prevention is the limitation of some of the potential concurrency. Static analysis of the source code, along with run-time analysis of the transaction and template attributes, can ensure deadlock prevention is activated only when needed.

For example, transactions that make all access requests in the same sequence do not need deadlock prevention. If static analysis can detect this pattern, or better, safely reorganize the code to ensure this pattern, deadlock prevention can be avoided. This assumes that the access permission is surrendered back to the pool of processes competing for access only when the transaction has no more need for the file pointer.

There are two areas where deadlock prevention can be extended. The first is in the definition of the transaction by a process. Currently, the definition of a transaction lies in the formal parameters of the remote function invocation. The creation of a transaction instance occurs when the remote process is passed a collection of file pointers. The scope of a transaction consists of the first access by any of the file pointers composing the transaction and the return of the remote function. Identification of any sub-transactions which would eliminate coupling between global file pointers reduces the need for deadlock prevention. Having only one global file pointer in a transaction eliminates the second condition for deadlock (Chapter 4.5).

The second extension for deadlock prevention occurs when a collective I/O operation takes place. The collective open must be taken into account by the deadlock prevention mechanism in the context of the current transaction of the process. PI/OT is designed to use a client-server model. Introducing a third process as the manager of the collective open (Chapter 4.2.1) complicates the deadlock prevention mechanism.

If there are any global file pointers in the current transaction, other than the one currently being collectively opened, the client process cannot request the open until access for the other pointers is confirmed. Since the process that is the manager of the collective open is not necessarily the same process that produced the current transaction, there are now three processes involved with the collective open. The client process ($\mathbf{P_C}$), the process that generated the transaction ($\mathbf{P_T}$), and the collective manager process ($\mathbf{P_M}$) must cooperate to prevent deadlock.

$\mathbf{P_C}$ cannot proceed to ask $\mathbf{P_M}$ for participation in the collective open until it has been granted access to the global file pointers in the transaction by $\mathbf{P_T}$. If the access to these file pointers has not been sought yet, which one should $\mathbf{P_C}$ use to ask for access? Recall that the client will be blocked until the access permission arrives. As well, $\mathbf{P_M}$ must ensure that it does not grant access to several client processes when more than one file is collectively opened. For example, two file pointers, **f** and **g**, with global semantics are collectively opened by processes $\mathbf{P_{C1}}$ and $\mathbf{P_{C2}}$. If $\mathbf{P_{C1}}$ is allowed to open **f**, the manager process ($\mathbf{P_M}$) must ensure that $\mathbf{P_{C1}}$ opens **g** before $\mathbf{P_{C2}}$'s requests for **f** or **g** can be granted. Otherwise, deadlock occurs.

Static analysis can be used to determine the scope of transactions. The transaction is currently defined by the scope of the remote function. However, within the function, sub-transactions can be defined (as evident by the effect of the early release function in Chapter 5.3). This subdivision must be kept under the control of the user since the user's algorithm can couple the two file pointers together despite the static analysis determining they are independent of one another.

6.1.2 Static Analysis Support

To solve the deadlock problems pointed out in the previous section and to increase the potential for concurrency, static analysis of the source code should provide hints to the run-time system. Also, this static analysis provides the chance to safely move I/O code within an application to reduce the length of time a transaction needs to run. Consider the case where a function reads in some data, computes, reads in some more data, computes, and then writes out the results. If it can be determined that the read and write operations are independent of one another, the static analysis could recommend that an early release be inserted after the second read operation. A further refinement of this transaction has the two read operations consolidated or clustered together at the beginning of the computations (if possible). After the read operations are completed, the file pointer is released. This allows another blocked process to read and start processing the data. The overall concurrency has been increased over the original solution. The first process to finish computing can then request access to the output file.

While the scope of a transaction is being defined in the remote function by the compiler, static analysis could be done in an attempt to automatically derive a segmentation function or constant.

6.1.3 Run-time Improvements

The run-time system makes decisions to grant access based on the current process configuration, the static analysis hints, and the **call-chain** list containing the current pending transactions. With more hints provided by the static analysis, concurrency could be safely improved.

One possible area of investigation is to examine the effectiveness of using on-demand paging for the cached file segments. In particular, when **unknown** segment sizes or a writable **photocopy** template are used, the current implementation reads from the starting point in the file to the end-of-file and sends that block of data as the cache. Instead, on-demand paging could cache only the parts that the remote process actually needs. The run-time system would need support from the operating system (or the run-time system would have to implement an equivalent paging system) to ensure that any meta-information associated with each page and the overall file is properly retrieved and updated. Possibly, a journal file approach would allow the original file to be updated as a series of I/O operations recorded by the remote process instead of overwriting the file or file segment by each returning process.

Using the append mode when opening a file gives the system the knowledge that there are no modifications to any existing data and that seeking backwards will not affect future changes. However, the file pointer still needs coordination and synchronization for writing. This leads to the situation where there are different views as to the location of the end-of-file, although reading and seeking backwards into the file will not need to be coordinated.

6.1.4 Extensions and Future Work Summary

Based on this prototype implementation of PI/OT, a number of future research directions have been suggested. The most important research direction involves static analysis of the application's source code. This analysis of the user's source code would identify sub-transactions, thus reducing the dependency on deadlock prevention and increasing the available concurrency for the application. Static analysis can be used to safely rearrange the code to optimize I/O access. In addition, hints for the run-time system to prefetch data can be introduced. Run-time refinements can take advantage of this improved knowledge base as well as using dynamic information such as opening a file using read-only or append mode to improve performance.

6.2 Contributions

Chapter 1.2 listed the proposed contributions of this work. The first contribution is to separate the parallel I/O specifications from the sequential functions using templates. That is, the goal was to keep the standard sequential interface for invoking I/O operations in the user's code and to describe **what** parallel I/O behaviour(s) are needed independent of the code. At compile and at run-time these specifications are used to identify and implement **how** the parallel behaviours will interact with the application and its environment.

The separation of parallel I/O specifications from the interface was accomplished by the five templates presented in Chapter 3. These simple behaviours can be composed to yield more complex I/O patterns. The composition of the simple behaviours, along with the read and write ordering attributes, allows the user greater flexibility in expressing the exact behaviour needed.

The second contribution of this work lies in the fact that by separating the I/O and computational parallelism from the sequential code, optimizations and adaptive behaviours at compile and run-time are possible. Developed in Chapters 3 and 4 and demonstrated in Chapter 5, the run-time system for I/O queries and cooperates with the computational parallelism to produce a concurrent application. When an application changes the I/O behaviour of a particular task type, the computational behaviour is not affected. However, the overall performance can be affected.

The third contribution is a validation that I/O templates do not necessarily imply a performance penalty. This claim is developed in Chapter 5. The software engineering advantages of templated I/O do not necessarily incur a corresponding loss of performance.

As a fourth contribution, this work identifies the components that interact between the computational and I/O parallel behaviours. There are three areas where interactions occurred when implementing PI/OT in Enterprise (Chapter 4.4). The first set of interactions defines a transaction by identifying parallel file pointers and determining the scope of these file descriptors using remote procedure call arguments (the computational parallelism) or collective opens (replication factors). In the second set of interactions, this defined transaction is used by the source-to-source translator to identify and modify I/O objects to reflect the desired parallel behaviours. Analysis will determine if deadlock prevention is needed. The third set of interactions is found in the run-time environment. When a transaction is activated, the run-time environment determines how the behaviours will be implemented. The location of file servers and processes, the number of processes participating in the transaction, and how the computational parallelism is implemented all have an effect on how the I/O parallelism is implemented.

The fifth and final contribution rests in the fact that this parallel I/O model is a step towards automatic parallelization. The abstraction mechanism for both the computational and I/O parallelism creates the opportunity for the user to specify **what** is wanted for the parallelism. Depending on the supplied code, the compiler can create a framework that at run-time can determine the best way (or **how**) to implement the parallelism.

6.3 Summary

This dissertation describes an attempt to define and implement a top-down model for parallel I/O. A user specifies **what** computational and I/O parallelism is wanted for an application. Static analysis and run-time support allows the application to determine **how** to implement the parallel behaviours. The central assumption is that the user develops code using standard C and the supplied standard stream interface (`stdio`). No new library of functions is needed to differentiate the parallel from sequential I/O operations. The cost of this approach does not mean significant loss of performance. Some compiler support is required to supply transaction information to the run-time system. This static analysis can be used to improve the concurrency through the insertion of an early release mechanism of file descriptors within a transaction and by determining if the deadlock prevention mechanism is necessary for a given transaction. The PI/OT parallel I/O model relies on an abstraction of the parallel computational model. Information is shared so that computational parallelism and I/O parallelism cooperatively work together to produce an efficient parallel application.